LINDO API 6.0

User Manual

LINDO Systems, Inc.

1415 North Dayton Street, Chicago, Illinois 60642
Phone: (312)988-7422 Fax: (312)988-9065
E-mail: info@lindo.com

COPYRIGHT

LINDO API and its related documentation are copyrighted. You may not copy the LINDO API software or related documentation except in the manner authorized in the related documentation or with the written permission of LINDO Systems, Inc.

TRADEMARKS

LINDO is a registered trademark of LINDO Systems, Inc. Other product and company names mentioned herein are the property of their respective owners.

DISCLAIMER

LINDO Systems, Inc. warrants that on the date of receipt of your payment, the disk enclosed in the disk envelope contains an accurate reproduction of LINDO API and that the copy of the related documentation is accurately reproduced. Due to the inherent complexity of computer programs and computer models, the LINDO API software may not be completely free of errors. You are advised to verify your answers before basing decisions on them. NEITHER LINDO SYSTEMS INC. NOR ANYONE ELSE ASSOCIATED IN THE CREATION, PRODUCTION, OR DISTRIBUTION OF THE LINDO SOFTWARE MAKES ANY OTHER EXPRESSED WARRANTIES REGARDING THE DISKS OR DOCUMENTATION AND MAKES NO WARRANTIES AT ALL, EITHER EXPRESSED OR IMPLIED, REGARDING THE LINDO API SOFTWARE, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR OTHERWISE. Further, LINDO Systems, Inc. reserves the right to revise this software and related documentation and make changes to the content hereof without obligation to notify any person of such revisions or changes.

Copyright ©2009 by LINDO Systems, Inc. All rights reserved.

Published by

LINDO SYSTEMS INC.

1415 North Dayton Street
Chicago, Illinois 60642
Technical Support: (312) 988-9421
E-mail: tech@lindo.com
http://www.lindo.com

TABLE OF CONTENTS

TABLE OF CONTENTS iii
Preface vii
Chapter 1: 1
Introduction 1
What Is LINDO API? 1
Linear Solvers 2
Mixed-Integer Solver 2
Nonlinear Solver 3
Global Solver 3
Stochastic Solver 3
Installation 3
Windows Platforms 4
Unix-Like Platforms 4
Updating License Keys 6
Solving Models from a File using Runlindo 7
Sample Applications 9
Array Representation of Models 9
Sparse Matrix Representation 10
Simple Programming Example 13
Chapter 2: 17
Function Definitions 17
Common Parameter Macro Definitions 18
Structure Creation and Deletion Routines 21
License and Version Information Routines 23
Input-Output Routines 25
Parameter Setting and Retrieving Routines 42
Available Parameters 53
Available Information 99
Model Loading Routines 114
Solver Initialization Routines 135
Optimization Routines 139
Solution Query Routines 144
Model Query Routines 160
Model Modification Routines 191
Model and Solution Analysis Routines 210
Error Handling Routines 219
Advanced Routines 221
Callback Management Routines 227
Memory Management Routines 238
Random Number Generation Routines 241
Sampling Routines 245
Chapter 3: 259
Solving Linear Programs 259
A Programming Example in C 259
A Programming Example in Visual Basic 269

VB and Delphi Specific Issues: ... 277
Chapter 4: Solving ... 279
Mixed-Integer Programs ... 279
Staffing Example Using Visual C++ ... 280
Staffing Example Using Visual Basic ... 287
Chapter 5: Solving Quadratic Programs ... 295
Setting up Quadratic Programs ... 296
Loading Quadratic Data via Extended MPS Format Files ... 296
Loading Quadratic Data via API Functions ... 297
Sample Portfolio Selection Problems ... 300
Example 1. The Markowitz Model: ... 300
Example 2. Portfolio Selection with Restrictions on the Number of Assets Invested: ... 304
Chapter 6: Solving Second-Order Cone Programs ... 311
Setting up Second-Order Cone Programs ... 314
Loading Cones via Extended MPS Format Files ... 314
Loading Cones via API Functions ... 316
Example 3: Minimization of Norms: ... 316
Converting Models to SOCP Form ... 321
Example 4: Ratios as SOCP Constraints: ... 322
Quadratic Programs as SOCP ... 326
Chapter 7: Solving Nonlinear Programs ... 329
Black-Box Style Interface ... 330
Loading Model Data ... 331
Evaluating Nonlinear Terms via Callback Functions ... 333
Instruction-List Style Interface ... 337
Postfix Notation in Representing Expressions ... 337
Supported Operators and Functions ... 339
Grey-Box Style Interface ... 346
Instruction Format ... 348
Example 1 ... 348
Example 2 ... 348
Example 3 ... 349
Differentiation ... 349
Solving Non-convex and Non-smooth models ... 350
Linearization ... 350
Multistart Scatter Search for Difficult Nonlinear Models ... 352
Global Optimization of Difficult Nonlinear Models ... 354
Sample Nonlinear Programming Problems ... 355
Example 1: Black-Box Style Interface: ... 355
Example 2: Instruction-List Style Interface ... 361
Example 3: Multistart Solver for Non-Convex Models ... 371
Example 4: Global Solver with MPI Input Format ... 375
Example 5: Grey-Box Style Interface ... 381
Chapter 8: ... 389
Stochastic Programming ... 389
Multistage Decision Making Under Uncertainty ... 389
Recourse Models ... 391
Scenario Tree ... 391
Setting up SP Models: ... 393
Loading Core Model: ... 394

Loading the Time Structure: ... 397
Loading the Stochastic Structure: ... 399
Monte Carlo Sampling ... 406
Sample SP Problems ... 411
An Investment Model to Fund College Education: ... 411
An American Put-Options Model: ... 413
Appendix 8a: Correlation Specification ... 415
Appendix 8b: Random Number Generation ... 415
Appendix 8c: Variance Reduction ... 416
Appendix 8d: The Costs of Uncertainty, EVMU and EVPI ... 416
Chapter 9: ... 421
Using Callback Functions ... 421
Specifying a Callback Function ... 421
A Callback Example Using C ... 424
A Callback Example Using Visual Basic ... 429
Integer Solution Callbacks ... 430
Chapter 10: Analyzing Models and Solutions ... 433
Sensitivity and Range Analysis of an LP ... 433
Diagnosis of Infeasible or Unbounded Models ... 435
Infeasible Models ... 435
Unbounded Linear Programs ... 437
Infeasible Integer Programs ... 438
Infeasible Nonlinear Programs ... 438
An Example for Debugging an Infeasible Linear Program ... 438
Block Structured Models ... 444
Determining Total Decomposition Structures ... 446
Determining Angular Structures ... 447
Chapter 11: mxLINDO ... 449
A MATLAB Interface ... 449
Introduction ... 449
Setting up MATLAB to Interface with LINDO ... 449
Using the mxLINDO Interface ... 450
Calling Conventions ... 452
mxLINDO Routines ... 452
Structure Creation and Deletion Routines ... 452
License Information Routines ... 455
Input-Output Routines ... 456
Error Handling Routines ... 464
Parameter Setting and Retrieving Routines ... 466
Model Loading Routines ... 473
Solver Initialization Routines ... 486
Optimization Routines ... 490
Solution Query Routines ... 491
Model Query Routines ... 498
Model Modification Routines ... 517
Model and Solution Analysis Routines ... 534
Advanced Routines ... 541
Callback Management Routines ... 546
Auxiliary Routines ... 552
Sample MATLAB Functions ... 554

M-functions using mxLINDO 554
Chapter 12: 557
An Interface to Ox 557
Introduction 557
Setting up Ox Interface 557
Calling Conventions 558
Example. Portfolio Selection with Restrictions on the Number of Assets Invested 560
Appendix A: Error Codes 565
Appendix B: 573
MPS File Format 573
Integer Variables 575
Semi-continuous Variables 576
SOS Sets 577
SOS2 Example 578
Quadratic Objective 579
Quadratic Constraints 580
Second Order Cone Constraints 581
Appendix C: 585
LINDO File Format 585
Flow of Control 585
Formatting 585
Optional Modeling Statements 587
FREE Statement 587
GIN Statement 588
INT Statement 588
SUB and SLB Statements 589
TITLE Statement 589
Appendix D: 591
MPI File Format 591
Appendix E: 593
SMPS File Format 593
CORE File 593
TIME File 593
STOCH File 595
Appendix F: 601
SMPI File Format 601
References 605
INDEX 607

Preface

LINDO Systems is proud to introduce LINDO API 6.0. The general features include a) stochastic optimization b) global and multistart solvers for global optimization, c) nonlinear solvers for general nonlinear optimization, d) simplex solvers for linear optimization e) barrier solvers for linear, quadratic and second-order-cone optimization f) mixed-integer solvers for linear-integer and nonlinear-integer optimization, g) tools for analysis of infeasible linear, integer and nonlinear models, h) interfaces to other systems such as MATLAB, Ox, Java and .NET and i) support of more platforms (see below). The primary solvers in LINDO API 6.0 are:

- **Stochastic Solver:**
  The stochastic programming solver provides the opportunity of decision making under uncertainty through multistage stochastic models with recourse. The user is required to express the uncertainity by providing distribution functions, either built-in or user-defined, and the stochastic solver will optimize the model to minimize the cost of the initial stage plus the expected value of recourse over the planning horizon. Advanced sampling modes are also available to approximate stochastic parameters from parametric distributions.

- **General Nonlinear Solver:**
  LINDO API is the first full-featured solver callable library to offer general nonlinear and nonlinear/integer capabilities. This unique feature allows developers to incorporate a single general purpose solver into their custom applications. As with its linear and integer capabilities, LINDO API provides the user with a comprehensive set of routines for formulating, solving, and modifying nonlinear models. The Nonlinear license option is required in order to use the nonlinear capabilities with LINDO API.

- **Global Solver:**
  The global solver combines a series of range bounding (e.g., interval analysis and convex analysis) and range reduction techniques (e.g., linear programming and constraint propagation) within a branch-and-bound framework to find proven global solutions to non-convex NLPs. Traditional nonlinear solvers can get stuck at suboptimal, local solutions. This is no longer the case when using the global solver.

- **Multistart Solver:**
  The multistart solver intelligently generates a sequence of candidate starting points in the solution space of NLP and mixed integer NLPs. A traditional NLP solver is called with each starting point to find a local optimum. For non-convex NLP models, the quality of the best solution found by the multistart solver tends to be superior to that of a single solution from a traditional nonlinear solver. A user adjustable parameter controls the maximum number of multistarts to be performed. See Chapter 7, *Solving Nonlinear Models*, for more information.

- **Barrier (Interior-Point) Solver:**
  Barrier solver is an alternative way for solving linear and quadratic programming problems. LINDO API's state-of-the-art implementation of the barrier method offers great speed advantages for large scale sparse models. LINDO API 6.0 also includes a special variant of the barrier solver specifically designed to solve *Second-Order-Cone* problems. See Chapter 6, *Solving Second-Order-Cone Models*, for more information.

- **Simplex Solvers:**

LINDO API 6.0 offers two advanced implementations of the primal and dual simplex methods as the primary means for solving linear programming problems. Its flexible design allows the users to fine tune each method by altering several of the algorithmic parameters.

- **Mixed Integer Solver:**
  The mixed integer solver's capabilities of LINDO API 6.0 extend to linear, quadratic, and general nonlinear integer models. It contains several advanced solution techniques such as a) cut generation b) tree reordering to reduce tree growth dynamically, and c) advanced heuristic and presolve strategies.

- **Model and Solution Analysis Tools:**
  LINDO API 6.0 includes a comprehensive set of analysis tools for a) debugging of infeasible linear, integer and nonlinear programs using series of advanced techniques to isolate the source of infeasibilities to smaller subset of the original constraints, b) performing sensitivity analysis to determine the sensitivity of the optimal basis to changes in certain data components (e.g. objective vector, right-hand-size values etc..).

- **Quadratic Recognition Tools:**
  The QP recognition tool is a useful algebraic pre-processor that automatically determines if an arbitrary NLP is actually a quadratic model. QP models may then be passed to the faster quadratic solver, which is available as part of the barrier solver option.

- **Linearization Tools:**
  Linearization is a comprehensive reformulation tool that automatically converts many non-smooth functions and operators (e.g., max and absolute value) to a series of linear, mathematically equivalent expressions. Many non-smooth models may be entirely linearized. This allows the linear solver to quickly find a global solution to what would have otherwise been an intractable nonlinear problem.

- **Decomposition Tools:**
  Many large scale linear and mixed integer problems have constraint matrices that are totally decomposable into a series of independent block structures. A user adjustable parameter can be set, so the solver checks if a model can be broken into smaller independent models. If total decomposition is possible, it will solve the independent problems sequentially to reach a solution for the original model. This may result in dramatic speed improvements. Refer to the *Block Structured Models* section in Chapter 10, *Analyzing Models and Solutions*, for more information.

- **Java Native Interface:**
  LINDO API includes Java Native Interface (JNI) support for Windows, Solaris, and Linux platforms. This new feature allows users to call LINDO API from Java applications, such as applets running from a browser.

- **MATLAB Interface:**
  The Matlab interface allows using LINDO API functions from within MATLAB. Using MATLAB's modeling and programming environment, you can build and solve linear, nonlinear, quadratic, and integer models and create custom algorithms based upon LINDO API's routines and solvers.

- **.NET Interface:**
  LINDO API includes C# and VB.NET interfaces that allow it to be used from within .NET's distributed computing environment (including Windows Forms, ADO.NET, and ASP.NET).

The interfaces are in the form of classes that allow managed .NET code to interact with unmanaged LINDO API code via the "System.Runtime.InteropServices" namespace.

- **Ox Interface:**
  This interface provides users of the Ox statistical package, the ability to call LINDO API's functions the same way they call native Ox functions. This offers greater flexibility in developing higher-level Ox routines that can set up and solve different kinds of large-scale optimization problems, testing new algorithmic ideas or expressing new solution techniques.

- **Platforms:**
  LINDO API 6.0 is currently available on Sparc Solaris 32/64 bit, Windows 32/64 bit, Linux 32/64-bit, Mac Intel 32-bit and Mac PowerPC 32-bit. For availability of LINDO API 6.0 on all other platforms, you may wish to contact LINDO Systems, Inc.

LINDO Systems, Inc
1415 N. Dayton
Chicago, Illinois
(312) 988 9421

info@lindo.com
http://www.lindo.com

January 2009

Chapter 1: Introduction

What Is LINDO API?

The LINDO Application Programming Interface (API) provides a means for software developers to incorporate optimization into their own application programs. LINDO API is designed to solve a wide range of optimization problems, including linear programs, mixed integer programs, quadratic programs, and general nonlinear non-convex programs. These problems arise in areas of business, industry, research, and government. Specific application areas where LINDO API has proven to be of great use include product distribution, ingredient blending, production and personnel scheduling, inventory management… The list could easily occupy the rest of this chapter.

Optimization helps you find the answer that yields the best result; attains the highest profits, output, or happiness; or achieves the lowest cost, waste, or discomfort. Often these problems involve making the most efficient use of your resources—including money, time, machinery, staff, inventory, and more. Optimization problems are often classified as linear or nonlinear, depending on whether the relationships in the problem are linear with respect to the variables.

The most fundamental type of optimization problems is the *linear program* (LP) of the form:

Minimize (or maximize) $c_1x_1 + c_2x_2 + \ldots + c_nx_n$

Such that

$$\begin{aligned} A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n \;\; ? \;\; b_1 \\ A_{21}x_1 + A_{22}x_2 + \ldots + A_{2n}x_n \;\; ? \;\; b_2 \\ \vdots \qquad\qquad \ldots \qquad\qquad \vdots \\ A_{m1}x_1 + A_{m2}x_2 + \ldots + A_{mn}x_n \;\; ? \;\; b_m \end{aligned}$$

$$\begin{aligned} L_1 \le x_1 \le U_1 \\ L_2 \le x_2 \le U_2 \\ \vdots \\ L_n \le x_n \le U_n \end{aligned}$$

where A_{ij}, c_j, b_i, L_j, U_j are known real numbers; *?* is one of the relational operators '≤', '=', or '≥'; and $x_1,x_2,\ldots,x_n$ are the decision variables (unknowns) for which optimal values are sought.

The expression being optimized is called the objective function and $c_1,c_2,\ldots,c_n$ are the objective coefficients. The relationships whose senses are expressed with *?* are the constraints; $A_{i1},A_{i2},\ldots,A_{in}$ are the coefficients; and b_i is the right-hand side value for the i^{th} constraint. L_j and U_j represent lower and upper bounds for the j^{th} decision variable and can be finite or infinite.

At the core of LINDO API's optimizers are the linear solvers, which solve problems of this form. On top of the linear solvers are other solver types. These can be used to solve generalizations of LPs, such as problems containing integer variables or quadratic or nonlinear expressions.

The problem of *mixed-integer linear programs* (MILP) is an extension of LPs where some of the decision variables are required to take integer (whole number) values. Another extension of LPs is when the expressions in the objective function or the constraints are nonlinear functions of decision variables, such as logarithms or products of variables. Such problems are called *nonlinear programs* (NLPs). A special case of NLPs is *quadratic programs* (QPs) where the only nonlinear relationships among variables are products of two variables. NLPs and QPs with integrality restrictions on some variables are called *mixed-integer nonlinear programs* (MINLP) and *mixed-integer quadratic programs* (MIQP), respectively.

Linear Solvers

There are three linear solvers—the Primal Simplex, Dual Simplex, and the Barrier Methods. The simplex method (primal or dual) solves the LP by moving along the edges of the feasible region defined by the constraint set. By contrast, the barrier method walks through the interior of the feasible region while searching an optimal solution. All three methods either terminate with an optimal solution or return a flag indicating that the LP is infeasible or unbounded.

In general, it is difficult to say which algorithm will be fastest for a particular model. A rough guideline is *Primal Simplex* tends to do better on sparse models with fewer rows than columns. *Dual Simplex* tends to do well on sparse models with fewer columns than rows or models that are primal and/or dual degenerate, while *Barrier* works best on structured models or very large models. The simplex methods use a state-of-the-art implementation of the revised simplex method with product form inverse. The barrier solver uses a homogeneous self-dual algorithm. All three use extensive preprocessing to help reduce the complexity of the LP and improve its numerical properties. See Chapter 3, *Solving Linear Programs*, for examples of solving linear programs with the LINDO API.

Mixed-Integer Solver

LINDO API solves the mixed-integer models with the branch-and-cut method. It is an iterative method that uses either the linear or nonlinear solver as a subsolver, depending on the nature of the problem. The mixed-integer solver is equipped with advanced preprocessing, heuristic and cut generation tools. Preprocessing generally reduces the problem size to a manageable size and offers great computational savings, especially for large problems. Addition of "cuts" helps eliminate the noninteger feasible regions quickly and provides improved bounds during the branch-and-bound. For many classes of MILP problems, heuristic algorithms quickly find good integer solutions and lead to improved bounds. All these techniques lead to improved solution times for most integer programming models. See Chapter 2, *Function Definitions*, for more information of optimization functions and related parameters. See Chapter 4, *Solving Mixed-integer Programs*, for examples of solving mixed integer programs with LINDO API.

Nonlinear Solver

LINDO API's nonlinear solver employs both successive linear programming (SLP) and generalized reduced gradient (GRG) methods. Under certain conditions, QPs, which are special cases of NLPs, can be solved more efficiently via the barrier method.

The nonlinear solver returns a local optimal solution to the underlying problem. If local optimality cannot be achieved, then a feasible solution is reported if one had been found. In case no feasible solutions were found or the problem was determined to be unbounded or numerical problems have been encountered, then an appropriate flag is returned.

LINDO API can automatically linearize a number of nonlinear relationships through the addition of constraints and integer variables, so the transformed linear model is mathematically equivalent to the original nonlinear model. Keep in mind, however, that each of these strategies will require additional computation time. Thus, formulating models, so they are convex and contain a single extremum, is desirable.

Global Solver

The standard nonlinear solver returns a local optimal solution to the NLP. However, many practical nonlinear models are non-convex and have more than one local optimal solution. In some applications, the user may want to find a global optimal solution.

The optional global solver available in LINDO API employs branch-and-cut methods to break an NLP model down into many convex sub-regions and returns a provably global optimal solution. See Chapter 7, *Solving Nonlinear Programs*, for examples of solving nonlinear programs with LINDO API.

LINDO API also has a multistart feature that restarts the standard (non-global) nonlinear solver from a number of intelligently generated points. This allows the solver to find a number of locally optimal points and report the best one found. This alternative could be used when global optimization is costly.

Stochastic Solver

LINDO API's stochastic solver can solve multistage linear, nonlinear and integer models where some of the model parameters are not known with certainity but can be expressed probabilistically. Integer and nonlinear stochastic models are solved by transforming the model into the so-called deterministic-equivalent model. Linear models can be solved either with the nested Benders method or through the deterministic equivalent. For models with parametric distributions, Monte-Carlo sampling is available for finite approximations. Standard variance reduction strategies like Latin-hypersquare sampling and antithetic control variates are also available during sampling. Advanced tools, like inducing a correlation structure among random parameters based on various measures, are also provided. See Chapter 8, *Stochastic Programming,* for a detailed coverage of the topic and illustrative examples.

Installation

Installing the LINDO API software is relatively straightforward. To run LINDO API, we recommend a computer running Linux, Solaris, or a 32-bit version of Windows. In general, you will need at least 32Mb of RAM and 20Mb of free disk space. A faster processor and additional memory may allow LINDO API to solve tougher problems and/or improve performance. It should be noted that these are minimums. Solving big models may require more resources.

Windows Platforms

To install a Windows version (95/98/NT/XP), simply insert the LINDO API installation CD, double-click on the LINDO API folder to open the directory, and then double-click on the setup icon to run the LINDO API setup program. Setup will do all the required work to install LINDO API on your system and will prompt you for any required information.

After the installation process is complete, the following directory structure will be available.

```
LINDOAPI\                  ' Installation directory
LINDOAPI\BIN\WIN32         ' Executables, dynamic libraries
LINDOAPI\LIB               ' Import library, Java class library
LINDOAPI\MATLAB            ' Matlab scripts, functions, etc..
LINDOAPI\OX                ' OX library
LINDOAPI\INCLUDE           ' Header files
LINDOAPI\LICENSE           ' License files
LINDOAPI\DOC               ' User manual in PDF format
LINDOAPI\SAMPLES           ' Samples directory
LINDOAPI\SAMPLES\C\        ' C/C++ samples
LINDOAPI\SAMPLES\DELPHI\   ' Delphi samples
LINDOAPI\SAMPLES\JAVA\     ' Java samples (J2SDK and J++)
LINDOAPI\SAMPLES\VB\       ' Visual Basic samples (Windows only)
LINDOAPI\SAMPLES\DOTNET\VB' Visual Basic .NET samples
LINDOAPI\SAMPLES\DOTNET\CS' C# .NET samples
LINDOAPI\SAMPLES\FORT\     ' F90 samples (Windows only)
LINDOAPI\SAMPLES\MPS\      ' Test problems in MPS format
```

Note: The binaries in your installation are located under 'lindoapi\bin\<platform>' directory, where <platform> refers to the platform (or operating system) you are working on. For instance, on x86 platform running 32-bit Windows, the binaries are located at 'lindoapi\bin\win32', similarly on x64 platform running 64-bit Linux, the binaries are at 'lindoapi\bin\linux64'.

Unix-Like Platforms

The installation is a little more involved on Unix/Linux platforms. Follow the steps below to complete the installation:

Step 1. Locate the LINDO API folder ('lindoapi/') on your installation CD. Copy this folder into a directory where you want to install the LINDO API (e.g., '/opt'). If you have downloaded LINDO API as a tarball (e.g., *LAPI-LINUX-IA32.tar.gz*), move the file into the following installation directory (32-bit Linux platform is assumed):

```
%> mv LAPI-LINUX-IA32.tar.gz /opt
```

and decompress it using GNU's tar utility as illustrated below:

```
/opt%> gtar -xzf LAPI-LINUX-IA32.tar.gz
```

This will create the LINDO API directory 'lindoapi/'. The directory structure of 'lindoapi/' will be similar to that on the Windows platform.

Note: Alternatively, in the absence of 'gtar' utility on your system, LINDO API can also be decompressed and extracted into your desired directory, using the following two steps:

```
/opt%> gzip –d LAPI-LINUX-IA32.tar.gz
/opt%> tar –xvf LAPI-LINUX-IA32.tar
```

Step 2. Change the working directory to 'lindoapi/bin/linux32' and check if LINDO API's shared libraries (.so files) and the driver program 'runlindo' are all in executable mode. If not, change their mode by typing the following:

```
%> chmod 755 liblindo.so.6.0
%> chmod 755 liblindojni.so.6.0
%> chmod 755 libmosek.so.6.0
%> chmod 755 libconsub3.so
%> chmod 755 runlindo
```

Step 3. Update the $LD_LIBRARY_PATH environment variable as follows (it is assumed that the installation directory is '/opt/lindoapi'):

```
LD_LIBRARY_PATH=/opt/lindoapi/bin/linux32:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

NOTE: Mac-OSX and AIX do not use LD_LIBRARY_PATH. Users of these systems should apply the following equivalent changes.

Mac-OSX:

```
DYLD_LIBRARY_PATH=/opt/lindoapi/bin/<platform>:$DYLD_LIBRARY_PATH
export DYLD_LIBRARY_PATH
```

AIX:

```
LIBPATH=/opt/lindoapi/bin/<platform>:$LIBPATH
export LIBPATH
```

On Linux platforms, you may prefer to include the '/opt/lindoapi/bin/linux32' directory into the '/etc/ld.so.conf' file and run 'ldconfig'. This operation generally requires super user privileges. You may also update the $PATH environment variable accordingly to include the '/opt/lindoapi/bin/<platform>' folder in your search path.

Step 4. Optionally, set $LINDOAPI_LICENSE_FILE environment variable to refer to the license file in your installation. For example, if your installation directory is /opt/lindoapi, then the environment variable should be set as

```
LINDOAPI_LICENSE_FILE = "/opt/lindoapi/license/lndapi60.lic"
export LINDOAPI_LICENSE_FILE
```

Alternatively, you can execute the shell script 'lindoapivars.sh' to perform the required updates in these environment variables. To execute this script manually, enter the following at command line

```
source /opt/lindoapi/bin/linux32/lindoapivars.sh
```

To execute this script automatically at logon, append this line to the end of your startup script (*.bashrc* or *.bash_profile* for bash shell).

Step 5. If you received a license file (lndapi60.lic) with your installation CD, copy this file into the '/opt/lindoapi/license' directory.

Step 6. You can test your installation by changing directory to /opt/lindoapi/bin/linux32 and type the following. This should display the version info on your LINDO API installation.

```
%>  ./runlindo –v
```

Optionally, you can add "." to your $PATH variable to avoid typing "./" in front of the program run from the current working directory.

Updating License Keys

In a default installation, the license file (lndapi60.lic) is located under LINDOAPI\LICENSE directory. The license file initially contains a demo license, which allows full access to LINDO API with limited problem capacity.

The contents of *lndapi60.lic* are as follows:

```
LINDO API
Demo
6.0
1
None
Nonlinear Global Barrier
Educational
All platforms
Eval Use Only
>
** place your license key here **
>
```

Modify this file by placing the license key you received with your copy of the software between the ">" characters. Be sure to preserve capitalization and include all hyphens. For instance, suppose your license key is: `AT3x-5*mX-6d9J-v$pG-TzAU-D2%`. Then, modify *lndapi60.lic*, so it appears *exactly* as follows:

```
LINDO API
Demo
6.0
1
None
Nonlinear Global Barrier
Educational
All platforms
Eval Use Only
>
AT3x-5*mX-6d9J-v$pG-TzAU-D2%
>
```

Note: If you purchased a license, you should update the license file with the license key you obtained from your sales representative.

If you were e-mailed your license key, simply cut the license key from the e-mail that contains it and paste it into the *lndapi60.lic* file replacing the existing demo license key.

Your license key is unique to your installation and contains information regarding your version's serial number, size, and supported options. The license key is case sensitive, so be sure to enter it exactly as

listed, including all hyphens. Given that your license key is unique to your installation, you can not share it with any user not licensed to use your copy of the LINDO API.

Solving Models from a File using Runlindo

LINDO API distribution contains a simple program, runlindo.exe that allows you to solve models from a file after installation. In a 32-bit Windows installation, runlindo.exe is in the \lindoapi\bin\win32 directory. Runlindo is intended to be run from a command line prompt. To see command line options, type at the command line prompt:

```
runlindo -help
```

in which case the following command line options will be listed:

```
Usage: RUNLINDO filename [ options ]

General options:
    -max          { Solve the problem as a maximization problem }
    -min          { Solve the problem as a minimization problem }
    -silent       { Do not log iterations to screen }
    -decomp       { Try decomposition when solving LP/MIP problems }
    -iisfind [n]  { Find IIS with search level 'n' }
    -iisnorm [n]  { Set infeasibility norm to 'n' in IIS search (1) }
    -iismeth [n]  { Use method 'n' with IIS finder (1) }
    -linfo        { Display license information }
    -ver,-v       { Display version and build date }
    -help,-h      { Help }
Linear optimization options:
    -lp           { Solve the problem as an LP problem }
    -psim         { Use the primal simplex method for LP problems }
    -dsim         { Use the dual simplex method for LP problems }
    -bar          { Use the barrier method for LP problems }
    -noscale      { Set scaling off }
    -dual         { Solve the dual model implicitly }
    -tpos         { Solve the dual model explicitly }
    -novertex     { No crossover with barrier method }
    -iusol        { Force the solver to return some solution
                    when the model is infeasible or unbounded }
    -pre_lp  [n]  { Set presolve level to 'n' for LP problems (126)}

Mixed integer optimization options:
    -mip          { Solve the problem as a MIP problem }
    -pri          { Read the priority file 'filename.ord' }
    -pre_root [n] { Set presolve level to 'n' for root node (510)}
    -pre_leaf [n] { Set presolve level to 'n' for leaf nodes (174)}
    -cut_root [n] { Set cut level to 'n' for root node (22526)}
    -cut_leaf [n] { Set cut level to 'n' for leaf nodes (20478)}
    -ord_tree [n] { Set tree reorder level to 'n' (10)}
    -heuris   [n] { Set heuristic level to 'n' (3)}
    -strongb  [n] { Set strongbranch level to 'n' (10)}
```

```
Nonlinear optimization options:
    -nlp         { Use the nonlinear solver for QP problems}
    -multis  [n] { Set number of multistarts to [n] for NLP problems}
    -conopt  [n] { Use conopt version 'n' for NLP problems (3)}
    -lnz     [n] { Set linearization for NLP problems to 'n' (0)}
    -pre_nlp [n] { Set presolve level to 'n' for NLP problems (0)}
    -dev     [n] { Set derivative type 'n' for NLP problems (0)}

Global optimization options:
    -gop         { Solve the problem as a GOP problem }

I/O options:
    -ini         { Read initial solution from 'filename.sol'}
    -sol         { Write solution to file 'filename.sol' }
    -sol_ipm     { Write IPM solution to file 'filename.sol' }
    -fmps        { Read formatted MPS files (old MPS format)}
    -cmps        { Read MPS compatible mode files
                   (can combine with -fmps) }
    -wmps        { Export the input model in MPS format }
    -wmpi        { Export the input model in MPI format }
    -wltx        { Export the input model in LINDO format }
    -wlng        { Export the input model in LINGO format }
    -wiis        { Export the IIS in LINDO format }
    -wset        { Export the input model with sets/sc in MPS format}
```

For example, to solve a linear program in MPS format in a file called “mymodel.mps”, you might type:

```
runlindo mymodel.mps  -sol
```

The option “-sol” causes a solution report to be written to the file “mymodel.sol”. To learn more about the file formats recognized, see the appendices.
The ability to set parameters is not limited to command line arguments. Before initializing each optimization session, runlindo reads optionally specified parameters from a file named “lindo.par”. All LINDO API parameters can be set through this simple interface. Parameter values set through command line arguments have precedence over those set through “lindo.par”. An example “lindo.par” can be found in:

```
lindoapi/bin/$PLATFORM
```

where $PLATFORM refers to one of the following

```
win32        for 32-bit MS Windows on x86,
win64        for 64-bit MS Windows on x64,
osx32x86     for 32-bit Macintosh OSX on x86
osx32ppc     for 32-bit Macintosh OSX on PowerPC
linux32      for 32-bit Linux on x86
linux64      for 64-bit Linux on x64
solaris32    for 32-bit Sun Solaris
solaris64    for 64-bit Sun Solaris
```

For details, on available parameters in LINDO API and their usage through API calls and parameter-files, see "Parameter Setting and Retrieving Routines" in Chapter 2."

Sample Applications

The distribution package contains several sample application programs that illustrate the use of LINDO API using a high level programming language. The majority of the examples provided are in C/C++. Sample applications in other languages, such as Visual Basic, C#, Delphi, Fortran 90, and Java/J++ are also given.

Note: The header files required by each programming language are located in *LINDOAPI\INCLUDE* directory. These headers contain macro definitions and function prototypes (calling sequences) for each programming language. For a detailed description of available LINDO API functions, please refer to Chapter 2, *Function Definitions*.

Each sample is located in a separate directory along with a MAKEFILE and/or an IDE Project (for Windows only) to build the application. Depending on your platform, use MAKEFILE.UNX (for Solaris and Linux) or MAKEFILE.WIN (for Windows).

Now, let's illustrate how to get started using LINDO API by setting up and solving a small LP using a programming language.

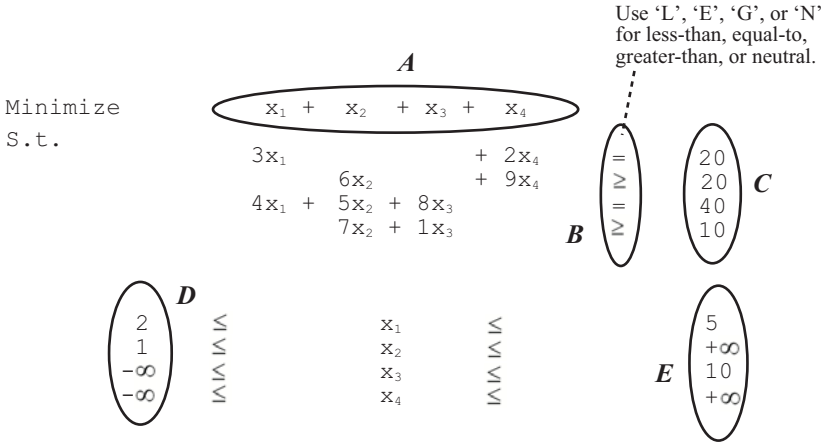
Array Representation of Models

From within a programming environment, models can be entered into LINDO API in either of two ways: 1) characterize the model using data structures (array representation) and pass the associated data objects to LINDO API via model loading routines in LINDO API, or 2) read the model from a file directly into LINDO API via input/output routines available. Supported file formats are MPS, LINDO, and MPI formats, which are described in Appendices *B*, *C*, and *D*, respectively. Here, we focus on the first alternative, which we have referred to as array representation, and describe how to characterize an LP model within a programming environment. In our discussion, the terms 'array' and 'vector' are used interchangeably.

We will use a small LP with four decision variables x_1, x_2, x_3, x_4 (n=4) and four constraints (m=4) for our example. The lower and upper bounds are specified for each variable explicitly. If neither bounds are given, it would be assumed the variable is continuous, bounded below by zero and bounded from above by infinity. The model appears as follows:

$$
\begin{array}{lrcrcl}
\text{Minimize} & & & x_1 + x_2 + x_3 + x_4 & & \\
\text{S.t.} & & & & & \\
 & 3x_1 & & + 2x_4 & = & 20 \\
 & & 6x_2 & + 9x_4 & \geq & 20 \\
 & 4x_1 + 5x_2 + 8x_3 & & & = & 40 \\
 & & 7x_2 + 1x_3 & & \geq & 10 \\
 & & & & & \\
2 & \leq & x_1 & & \leq & 5 \\
1 & \leq & x_2 & & \leq & +\infty \\
-\infty & \leq & x_3 & & \leq & 10 \\
-\infty & \leq & x_4 & & \leq & +\infty
\end{array}
$$

The diagram below shows how each component of LP data, except the coefficients of the constraint matrix, can be trivially represented by vectors (arrays). The circled elements labeled *A,B,C,D*, and *E* in the following figure symbolize these components and refer to *objective coefficients*, *constraint senses*, *right-hand sides*, *lower-bounds*, and *upper-bounds*, respectively.



In this small example, these vectors translate to the following:

```
A = [ 1   1   1   1 ].
B = [ E   G   E   G ].
C = [ 20  20  40  10 ].
D = [ 2   1   -LS_INFINITY   -LS_INFINITY ].
E = [ 5   LS_INFINITY   10   LS_INFINITY ].
```

Each of these vectors can be represented with an array of appropriate type and passed to LINDO API via model loading routines. Although it is also possible to represent the coefficients of the constraint matrix with a single vector, a different representation, called the *sparse matrix representation*, has been adopted. This is discussed in more detail below.

Sparse Matrix Representation

LINDO API uses a sparse matrix representation to store the coefficient matrix of your model. It represents the matrix using three (or optionally four) vectors. This scheme is utilized, so it is unnecessary to store zero coefficients. Given that most matrix coefficients in real world math programming models are zero, this storage scheme proves to be very efficient and can drastically reduce storage requirements. Below is a brief explanation of the representation scheme.

We will use the coefficients of the constraint matrix in our sample LP from above. These are as follows:

$$\begin{matrix} x_1 & x_2 & x_3 & x_4 \end{matrix}$$
$$\begin{bmatrix} 3 & 0 & 0 & 2 \\ 0 & 6 & 0 & 9 \\ 4 & 5 & 8 & 0 \\ 0 & 7 & 1 & 0 \end{bmatrix}$$

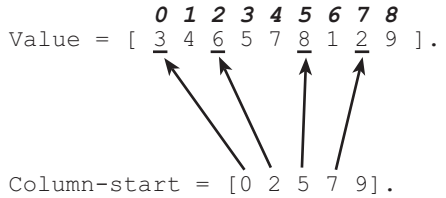
Three Vector Representation

Three vectors can represent a sparse matrix in the following way. One vector will contain all of the nonzero entries from the matrix, ordered by column. This is referred to as the *Value* vector. In our example, this vector has 9 entries and looks like:

```
Value = [ 3  4  6  5  7  8  1  2  9 ].
```

Note that all of the entries from the first column appear first, then the entries from the second column, and so on. All of the zeros have been stripped out.

In the second vector, which we call the *Column-start* vector, we record which points in the *Value* vector represent the start of a new column from the original matrix. The n^{th} entry in the *Column-start* vector tells us where in the *Value* vector to find the beginning of the n^{th} column. For instance, the column starts for the *Value* vector of our small example are underlined in the following diagram. Note that LINDO API uses zero-based counting, so the *Column-start* vector is as follows:



Note that the *Column-start* vector has one more entry than there are columns in our matrix. The extra entry tells LINDO where the last column ends. It will always be equal to the length of the *Value* vector.

From the *Column-start* vector, we can deduce which column is associated with each entry in our *Value* vector. The only additional information that we need is the row numbers of the entries. We store this information in a third vector, the *Row-index* vector. This vector is the same length as the *Value* vector. Each entry in the *Row-index* vector tells which row the corresponding entry from the *Value* vector belongs to. In our example, the number 3 belongs to the first row, which we call row 0, so the first entry in the *Row-index* vector is 0. Similarly, the second entry in the *Value* vector (4), belongs to the third row (row 2 when starting from zero), so the second entry of the *Row-index* vector is 2. Continuing in this way through the rest of the entries of the *Value* vector, the resulting *Row-index* vector appears as follows:

```
                0  1  2  3  4  5  6  7  8
Row-index = [ 0  2  1  2  3  2  3  0  1 ].
```

In summary, our transformation from a matrix into 3 vectors is:

$$\begin{bmatrix} 3 & 0 & 0 & 2 \\ 0 & 6 & 0 & 9 \\ 4 & 5 & 8 & 0 \\ 0 & 7 & 1 & 0 \end{bmatrix} \Rightarrow \begin{array}{ll} \text{Column-starts:} & [\,0\ 2\ 5\ 7\ 9\,] \\ \text{Value:} & [\,3\ 4\ 6\ 5\ 7\ 8\ 1\ 2\ 9\,] \\ \text{Row-index:} & [\,0\ 2\ 1\ 2\ 3\ 2\ 3\ 0\ 1\,] \end{array}$$

Four Vector Representation

The four vector representation allows more flexibility than the three vector representation. Use it when you expect to add rows to your original matrix (i.e., if you will be adding additional constraints to your model).

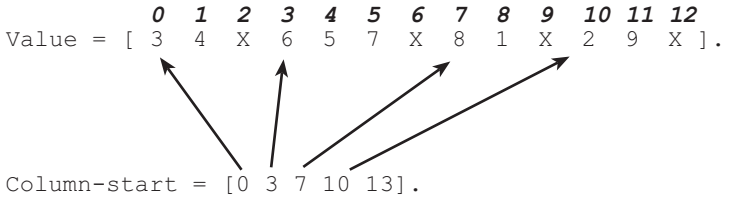
The four vector representation uses the same three vectors as above. However, it allows you to have "blanks" in your *Value* vector. Because of this, you must also pass a vector of column lengths, since the solver doesn't know how many blanks there will be.

For example, suppose we wish to leave room for one additional row. Then, our *Value* vector becomes:

```
Value = [ 3  4  X  6  5  7  X  8  1  X  2  9  X ]
```

where the *X*'s represent the blanks. The blanks may be nulls or any other value, since they will be ignored for the time being.

Our *Column-start* vector becomes:



Our new vector is the *Column-length* vector. It will contain the length of each column (i.e., the number of nonzeros in each column). This allows the solver to skip the blanks (*X*'s) in the *Value* vector. In our small example, since the first column contains two nonzero and nonblank entries, the first element of the *Column-length* vector will be 2. Continuing through the remaining columns, the *Column-length* vector and its corresponding entries from the *Value* vector are as follows:

```
Column-length = [ 2  3  2  2 ].

Value = [ 3  4  X  6  5  7  X  8  1  X  2  9  X ].
```

Our *Row-index* vector is as before, except we add a blank for each blank in the *Value* vector. As with the *Value* vector, these blanks will be ignored, so they can contain any value. Thus, the *Row-index* vector becomes:

```
                0  1  2  3  4  5  6  7  8  9  10 11 12
Row-index = [ 0  2  X  1  2  3  X  2  3  X  1  2  X ].
```

In summary, the four vector transformation is:

$$\begin{bmatrix} 3 & 0 & 0 & 2 \\ 0 & 6 & 0 & 9 \\ 4 & 5 & 8 & 0 \\ 0 & 7 & 1 & 0 \end{bmatrix} \Rightarrow$$

Column lengths: [2 3 2 2]
Column starts: [0 3 7 10 13]
Values: [3 4 X 6 5 7 X 8 1 X 2 9 X]
Row indexes: [0 2 X 1 2 3 X 2 3 X 0 1 X]

Simple Programming Example

Up to this point, we have seen that the objective function coefficients, right-hand side values, constraint senses, and variable bounds can be stored in vectors of appropriate dimensions and the constraint matrix can be stored in three or four vectors using the sparse matrix representation. In this section, we show how these objects should be declared, assigned values, and passed to LINDO API to complete the model setup phase and invoke optimization.

Recall the small LP example model from the array representation section above:

$$
\begin{aligned}
\text{Minimize} \quad & x_1 + x_2 + x_3 + x_4 \\
\text{S.t.} \quad & \\
& 3x_1 + 2x_4 = 20 \\
& 6x_2 + 9x_4 \geq 20 \\
& 4x_1 + 5x_2 + 8x_3 = 40 \\
& 7x_2 + 1x_3 \geq 10 \\
& 2 \leq x_1 \leq 5 \\
& 1 \leq x_2 \leq +\infty \\
& -\infty \leq x_3 \leq 10 \\
& -\infty \leq x_4 \leq +\infty
\end{aligned}
$$

It is easy to verify that the model has 4 variables, 4 constraints, and 7 nonzeros. As determined in the previous section, its constraint matrix has the following (three-vector) sparse representation:

```
Column-start = [ 0  2  5  7  9 ]
Values       = [ 3.0  4.0  6.0  5.0  7.0  8.0  1.0  2.0  9.0 ]
Row-index    = [ 0  2  1  2  3  2  3  0  1 ]
```

Other components of the LP data, as described above, are:

```
Right-hand side values = [ 20  20  40  10 ].
Objective coefficients = [ 1  1  1  1 ].
Constraint senses = [ E  G  E  G ].
Lower bounds = [ 2  1  -LS_INFINITY  -LS_INFINITY ].
Upper bounds = [ 5  LS_INFINITY  10  LS_INFINITY ].
```

Create an Environment and Model

Before any data can be input to LINDO API, it is necessary to request LINDO API to initialize the internal solvers by checking the license this user has and to get handles of the required resources (e.g., pointers to internal memory areas). This is achieved by creating a LINDO environment object and creating a model object within the environment. These reside at the highest level of LINDO API's internal object oriented data structure. In this structure, a model object belongs to exactly one environment object. An environment object may contain zero or more model objects.

The following code segment does this:

```
/* declare an environment variable */
pLSenv pEnv;

/* declare a model variable */
pLSmodel pModel;

/* Create the environment./
pEnv = LScreateEnv ( &nErrorCode, MY_LICENSE_KEY);

/* Create the model./
pModel = LScreateModel  ( pEnv, &nErrorCode);
```

The environment data type, *pLSenv*, and the model data type, *pLSmodel*, are both defined in the *lindo.h* header file. A call to *LScreateEnv()* creates the LINDO environment. Finally, the model object is created with a call to *LScreateModel()*. For languages other than C/C++ *pLSenv* and *pLSmodel* objects refer to integer types. The associated header files are located in the 'lindoapi/include' directory.

Load the Model

The next step is to set up the LP data and load it to LINDO API. This is generally the most involved of the steps.

Objective

The following code segment is used to enter the direction of the objective. The possible values for the direction of the objective are *LS_MAX* and *LS_MIN*, which are predefined macros that stand for maximize or minimize. For our sample problem, the objective direction is given as maximization with the following code:

```
int nDir = LS_MIN;
```

The constant terms in the objective function are stored in a double scalar with the following:

```
double dObjConst = 0.0;
```

Finally, the objective coefficients are placed into an array with the following:

```
double adC[4] = { 1., 1., 1., 1.};
```

Constraints

The following code segment is used to enter the number of constraints:

```
int nM = 4;
```

The constraint right-hand sides are place into an array with the following:

```
double adB[4] = { 20., 20., 40., 10. };
```

The constraint types are placed into an array with the following:

```
char acConTypes[4] = {'E', 'G', 'E' , 'G' };
```

The number of nonzero coefficients in the constraint matrix is stored:

```
int nNZ = 9;
```

Finally, the length of each column in the constraint matrix is defined. This is set to NULL in this example, since no blanks are being left in the matrix:

```
int *pnLenCol = NULL;
```

The nonzero coefficients, column-start indices, and the row indices of the nonzero coefficients are put into arrays with the following:

```
int anBegCol[5] = { 0  , 2  , 5  , 7  , 9 };
double   adA[9] = { 3.0, 4.0, 6.0, 5.0, 7.0, 8.0, 1.0, 2.0, 9.0 };
int   anRowX[9] = { 0  , 2  , 1  , 2  , 3  , 2  , 3  , 0  , 1   };
```

Note:	Refer to the section *Sparse Matrix Representation* above for more information on representing a matrix with three or four vectors.

Variables

The following code segment is used to declare the number of variables:

```
int nN = 4;
```

The upper and lower bounds on the variables are defined with the following:

```
double pdLower[4] = {2, 1, -LS_INFINITY, -LS_INFINITY};
double pdUpper[4] = {5, LS_INFINITY, 10, LS_INFINITY};
```

Then, the variable types are placed into an array with the following:

```
char acVarTypes[4] = {'C','C','C','C'};
```

The variable types could actually be omitted and LINDO API would assume that the variables were continuous.

We have now assembled a full description of the model and pass this information to LINDO API with the following:

```
nErrorCode = LSloadLPData( pModel, nM, nN, nDir, dObjConst, adC, adB,
acConTypes, nNZ, anBegCol, pnLenCol, adA, anRowX, pdLower, pdUpper);
```

All LINDO API functions return an error code indicating whether the call was successful or not. If the call was successful, then the error code is zero. Otherwise, an error has occurred and its type could be looked up in Appendix A, *Error Codes*. It is imperative that the error code returned is always checked to verify that the call was successful.

Note: If there is a nonzero error code, the application program should stop, since the results would be unpredictable and it may cause the program to crash.

Solve

Since the model is an LP, a linear solver, such as the primal simplex method, can be used. The model is solved with the following call:

```
nErrorCode = LSoptimize( pModel, LS_METHOD_PSIMPLEX, &nSolStat);
```

Alternative solvers available for linear models include dual simplex and barrier (if licensed). When the second argument in the function call is set to LS_METHOD_FREE, LINDO API will decide the solver to use by examining its structure and mathematical content. See the *Common Macro Definitions* section of Chapter 2, *Function Definitions*, for more information on the predefined macros LS_METHOD_PSIMPLEX and LS_METHOD_FREE.

Retrieve the Solution

The next step is to retrieve the solution using solution query functions. Many of the LINDO API query functions need to have space allocated before calling the routine. You must be sure to allocate sufficient space for query routines that include a pointer to a string, an integer vector, a double precision vector, or character vector. If sufficient memory is not initially allocated, the application will crash once it is built and executed. See *Solution Query Routines* in Chapter 2, *Function Definitions*, for more information on which routines require space to be allocated for them. Refer to Chapter 3, *Solving Linear Programs*, for more details on building and solving the model and a programming example in Visual Basic.

Here, the objective value and optimal variable values will be displayed. The objective value is retrieved and printed with the following:

```
double adX[4];
nErrorCode = LSgetInfo( pModel, LS_DINFO_POBJ, &dObj);
printf( "Objective Value = %g\n", dObj);
```

See the context of the *LSgetInfo()* function in Chapter 2, *Function Definitions*, for more information on the predefined macro LS_DINFO_POBJ. It tells LINDO API to fetch the value of the primal objective value via the *LSgetInfo()* function. The optimal variable values are retrieved and printed with the following:

```
nErrorCode = LSgetPrimalSolution ( pModel, adX);
printf ("Primal values \n");
for (i = 0; i < nN; i++) printf( " x[%d] = %g\n", i, adX[i]);
printf ("\n");
```

The output of this program would appear as follows:

```
Objective Value = 10.44118
Primal values
 x[0] = 5
 x[1] = 1.176471
 x[2] = 1.764706
 x[3] = 2.5
```

Clear Memory

A last step is to release the LINDO API memory by deleting the LINDO environment with the following call:

```
nErrorCode = LSdeleteEnv( &pEnv);
```

This frees up all data structures LINDO API allocated to the environment and all of the environment's associated models.

Chapter 2: Function Definitions

In this section, we give "header" definitions of all user callable functions in LINDO API. Most of the functions return error or information codes. For a complete listing of the codes, see Appendix A, *Error Codes.*

The general form of functions in LINDO API is:

LSverbObject(*specific_object*)

Typical verbs are: create, delete, get, set, and optimize. Typical objects are: environment, model, and parameter. LINDO API assumes typical default values (e.g., zero or infinity (LS_INFINITY) for most of the specific objects). If you are happy with these defaults, then you can simply pass NULL for these arguments in a function call.

In describing the callable functions, we have adopted a variable-naming convention, which is commonly referred to as *Hungarian notation*. Several versions of Hungarian notation have evolved over the years and all its dialects are intended to be *mnemonic* (easy to remember) for your convenience. The version used here is dictated mainly by the data structure definitions that arise in the representation of mathematical models. In building your own applications, you may follow or alter them as desired.

In Hungarian notation, variable names begin with one or more lowercase letters that denote the variable type, thus providing an inherent identification. For example, the prefix *ad* is used to identify a double precision array, as in *adVal***.** In like fashion, the prefix *psz* identifies a pointer to a null-terminated string, as in ***pszFilename***. The following table summarizes the Hungarian notation conventions for the commonly used data types in LINDO API:

Prefix	**Data type**
a	Array
c	Integer (count)
ch	Character
d	Double
f	Float
i	Integer (index into arrays)
n	Integer
p	A pointer variable containing the address of a variable
sz	Null-terminated string (ASCIIZ)

Common Parameter Macro Definitions

Macro symbols are extensively used in LINDO API as arguments during function calls or as a parameter to specify a choice or value. The macros can take integer, double, or character values. In the following table, the most common ones are listed. The first column gives the name of the macro, the second column refers to the value it refers to, and the third column gives a brief description.

Symbol	Value	Description
Model Status		
LS_STATUS_OPTIMAL	1	An optimal solution is found
LS_STATUS_BASIC_OPTIMAL	2	An optimal basic solution is found
LS_STATUS_INFEASIBLE	3	The model is infeasible
LS_STATUS_UNBOUNDED	4	The model is unbounded
LS_STATUS_FEASIBLE	5	The model is feasible
LS_STATUS_INFORUNB	6	The solution is infeasible or unbounded. In order to determine the actual status, primal simplex method should be run on the model with presolver off.
LS_STATUS_NEAR_OPTIMAL	7	A near optimal solution is found (for nonlinear problems only)
LS_STATUS_LOCAL_OPTIMAL	8	A local optimal solution is found (for nonlinear problems only)
LS_STATUS_LOCAL_INFEASIBLE	9	A locally infeasible solution is found (for nonlinear problems only)
LS_STATUS_CUTOFF	10	The solver found an optimal solution worse than the cutoff
LS_STATUS_NUMERICAL_ERROR	11	The solver encountered a numerical error during a function evaluation (e.g., square root of a negative number)
LS_STATUS_UNKNOWN	12	Model was attempted to be solved, but the optimization session terminated without producing any useful information as to what the actual status of the model is. So, the status of the model is remains unknown.
LS_STATUS_UNLOADED	13	No model is loaded
LS_STATUS_LOADED	14	Model is loaded, but it has not been attempted to be solved yet.
Optimization Direction		
LS_MIN	1	Minimization type model.
LS_MAX	-1	Maximization type model.
Numerical Infinity		
LS_INFINITY	1.E30	Numeric infinity for variable bounds. All bounds whose absolute value is larger than

		LS_INFINITY is truncated.
Constraint Types (Senses)		
LS_CONTYPE_LE	'L'	Less than equal to.
LS_CONTYPE_EQ	'E'	Equal to.
LS_CONTYPE_GE	'G'	Greater than equal to.
LS_CONTYPE_FR	'N'	Free (or neutral).
Cone Types		
LS_CONETYPE_QUAD	'Q'	Quadratic cone
LS_CONETYPE_RQUAD	'R'	Rotated quadratic cone
Variable Types		
LS_VARTYPE_CONT	'C'	Continuous variable.
LS_VARTYPE_BIN	'B'	Binary variable.
LS_VARTYPE_INT	'I'	General integer variable.
Solver Types		
LS_METHOD_FREE	0	Solver decides.
LS_METHOD_PSIMPLEX	1	Primal simplex method.
LS_METHOD_DSIMPLEX	2	Dual simplex method.
LS_METHOD_BARRIER	3	Barrier method.
LS_METHOD_NLP	4	Nonlinear Solver.
Basis Status		
LS_BASTYPE_BAS	0	Basic.
LS_BASTYPE_ATLO	-1	Non-basic at lower bound.
LS_BASTYPE_ATUP	-2	Non-basic at upper bound.
LS_BASTYPE_FNUL	-3	Free and non-basic at zero value.
LS_BASTYPE_SBAS	-4	Fixed and non-basic at both lower and upper bounds.
Solution File Format and Types		
LS_SOLUTION_OPT	0	Default solution file format.
LS_SOLUTION_MIP	1	Solution file format for MIP solutions.
LS_SOLUTION_OPT_IPM	2	Solution file format for interior point solutions.
LS_SOLUTION_OPT_OLD	3	Solution file format in LINDO API version 1.x.
LS_SOLUTION_MIP_OLD	4	Solution file format for MIP solutions in LINDO API version 1.x
Set Types		
LS_MIP_SET_SOS1	1	Special ordered set of type-1
LS_MIP_SET_SOS2	2	Special ordered set of type-2
LS_MIP_SET_SOS3	3	Special ordered set of type-3

LS_MIP_SET_CARD	4	Set cardinality.
Norm Options		
LS_IIS_NORM_FREE	0	Solver decides the infeasibility norm for IIS analysis.
LS_IIS_NORM_ONE	1	Solver uses L-1 norm for IIS analysis.
LS_IIS_NORM_INFINITY	2	Solver uses L-∞ norm for IIS analysis
IIS Methods		
LS_IIS_DEFAULT	0	Use default filter in IIS analysis.
LS_IIS_DEL_FILTER	1	Use deletion filter in IIS analysis.
LS_IIS_ADD_FILTER	2	Use additive filter in IIS analysis.
LS_IIS_GBS_FILTER	3	Use generalized-binary-search filter in IIS analysis.
LS_IIS_DFBS_FILTER	4	Use depth-first-binary-search filter in IIS analysis.
LS_IIS_FSC_FILTER	5	Use fast-scan filter in IIS analysis.
LS_IIS_ELS_FILTER	6	Use elastic filter in IIS analysis.
Stochastic Optimization Methods		
LS_METHOD_STOC_FREE	-1	Solve with the method chosen by the solver.
LS_METHOD_STOC_DETEQ	0	Solve the deterministic equivalent (DETEQ).
LS_METHOD_STOC_NBD	1	Solve with the Nested Benders Decomposition (NBD) method.
LS_METHOD_STOC_ALD	2	Solve with the Augmented Lagrangian Decomposition (ALD) method.
Stochastic Data Types		
LS_JCOL_INST	-8	Stochastic parameter is an instruction code
LS_JCOL_RUB	-7	Stochastic parameter is an upper bound for RHS (reserved for future use)
LS_JCOL_RLB	-6	Stochastic parameter is a lower bound for RHS (reserved for future use)
LS_JCOL_RHS	-5	Stochastic parameter is a RHS value (belongs to RHS column)
LS_IROW_OBJ	-4	Stochastic parameter is an objective coefficient (belongs to OBJ row)
LS_IROW_VUB	-3	Stochastic parameter is a lower bound (belongs to LO row)
LS_IROW_VLB	-2	Stochastic parameter is an upper bound (belongs to UP row)
LS_IROW_VFX	-1	Stochastic parameter is a fixed bound (belongs to FX row)
LS_IMAT_AIJ	0	Stochastic parameter is an LP matrix entry.

Structure Creation and Deletion Routines

The routines in this section are used to create and destroy the basic data structures used within LINDO API to manage your mathematical programming models.

In order to solve a model, you must begin by allocating a modeling environment. This is done through a call to *LScreateEnv*(). LINDO API uses the environment space to store global data pertaining to all models belonging to the environment. Once an environment has been created, you allocate space for one or more models within the environment. Models are allocated by calls to *LScreateModel*(). The model structure holds all model specific data and parameters.

LScreateEnv()

Description:

Creates a new instance of *LSenv*, which is an environment used to maintain zero or more models. The *LSenv* data structure is defined in the *lindo.h* header file.

Returns:

If successful, a pointer to the newly created instance of *LSenv* is returned. If unsuccessful, NULL is returned.

Prototype:

pLSenv	LScreateEnv(int *pnErrorcode, char *pszPassword)

Input Arguments:

Name	Description
pszPassword	A pointer to a character string containing a license key for LINDO API.

Output Arguments:

Name	Description
pnErrorcode	A pointer to the error code. If successful, *pnErrorcode will be 0 on return. A list of possible error codes may be found in Appendix A, *Error Codes*.

Remarks:

- Your license key is printed on the sleeve containing the distribution CD.
- You can call *LSloadLicenseString*() to read the license key from a text file.
- Be sure to call *LSdeleteEnv* (see below) once for each environment created when they are no longer needed. This will allow LINDO API to free all memory allocated to the environments.

LScreateModel()

Description:

Creates a new instance of *LSmodel*.

Returns:

If successful, a pointer to the newly created instance of *LSmodel* is returned. If unsuccessful, NULL is returned.

Prototype:

pLSmodel	LScreateModel(pLSenv pEnv, int *pnErrorcode)

Input Arguments:

Name	Description
pEnv	Pointer to the current LINDO environment established via a call to *LScreateEnv*().

Output Arguments:

Name	Description
pnErrorcode	A pointer to the error code. If successful, *pnErrorcode will be 0 on return. A list of potential error codes is listed in Appendix A, *Error Codes.*

Remarks:

- *LScreateEnv*() must be called before this function is called in order to obtain a valid environment pointer.
- Be sure to call *LSdeleteModel*() (see below) once for each model created when they are no longer needed. This will allow LINDO API to free all memory allocated to the models.

LSdeleteEnv()

Description:

Deletes an instance of *LSenv*. The memory used by the *LSenv* instance is freed and the pointer to the instance is set to NULL. Each model created under this environment will also be deleted by calls to *LSdeleteModel*().

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSdeleteEnv(pLSenv *pEnv)

Input Arguments:

Name	Description
pEnv	A pointer to a pointer of an instance of *LSenv*.

LSdeleteModel()

Description:

Deletes an instance of *LSmodel*. The memory used by the *LSmodel* instance is freed and the pointer to this instance is set to NULL.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSdeleteModel(pLSmodel *pModel)

Input Arguments:

Name	Description
pModel	A pointer to a pointer of an instance of *LSmodel*.

License and Version Information Routines

The first routine in this section allows you to read a license key from a file and load it into a local string buffer. Your license key is unique to your installation and contains information regarding your version's serial number, size, and supported options. The license key is case sensitive, so be sure to enter it exactly as listed, including all hyphens. Given that your license key is unique to your installation, you should not share it with any user not licensed to use your copy of LINDO API. The second routine allows you to access the version and build date of LINDO API.

LSgetVersionInfo()

Description:

Returns the version and build information of the LINDO API on your system.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetVersionInfo(char *pszVersion, char *pszBuildData)

Output Arguments:

Name	Description
pszVersion	A pointer to a null terminated string that keeps the version information of the LINDO API on your system.
pszBuildDate	A pointer to a null terminated string that keeps the build date of the LINDO API library on your system.

LSloadLicenseString()

Description:

Reads the license string from the specified file in text format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadLicenseString(char *pszFname, char *pszLicense)

Input Arguments:

Name	Description
pszFname	A pointer to a null terminated string that refers to the name of the file that contains your license key. Typically, the license key is placed in the *lndapi60.lic* file.

Output Arguments:

Name	Description
pszLicense	A pointer to a null terminated string that keeps the license key.

Input-Output Routines

The routines in this section provide functionality for reading and writing model formulations to and from disk files. Loading a model from a file will generally not be as efficient as passing the nonzero structure directly via the routines discussed in the *Model Loading Routines* section below. However, some may find files more convenient.

LINDO API currently supports four file formats: LINDO, MPS, LINGO, and MPI. LINDO format is identical to the format used by the interactive version of LINDO and is very straightforward to use. The LINDO format is discussed in detail in Appendix C, *LINDO File Format*. MPS format, although not as easy to deal with as LINDO format, is an industry standard and can be processed by most commercial solvers. The details of the MPS format are given in Appendix B, *MPS File Format*. The LINGO format is similar to the LINDO format and was originally developed for use with the LINGO modeling language. For details on the LINGO format, refer to the *LINGO User's Manual*, available through LINDO Systems. MPI format is for representing nonlinear models, which is described in detail in Appendix D, *MPI File Format*. LINDO API can read and write both LINDO and MPS files. At present, LINGO files may only be written and may not be read, and MPI files can only be read.

LSreadLINDOFile()

Description:

Reads the model in LINDO format from the given file and stores the problem data in the given model structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSreadLINDOFile(pLSmodel pModel, char *pszFname)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the model. To obtain a pointer to a model structure, see *LScreateModel*().
pszFname	A pointer to a null terminated string containing the path and name of the LINDO file.

Remarks:

- Details for the LP file format are given in Appendix C, *LINDO File Format*.
- To write a model in LINDO format, see *LSwriteLINDOFile*().
- To read a model in MPS format, see *LSreadMPSFile*().

LSreadMPSFile()

Description:

Reads a model in MPS format from the given file and stores the problem data in the given problem structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSreadMPSFile(pLSmodel pModel, char *pszFname, int nFormat)

Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the model. To obtain a pointer to a model structure, see *LScreateModel*().
pszFname	A pointer to a null terminated string containing the path and name of the MPS file.
nFormat	An integer parameter indicating whether the MPS file is formatted or not. The parameter value should be either LS_FORMATTED_MPS or LS_UNFORMATTED_MPS.

Remarks:

- All memory for the passed *LSmodel* structure will be allocated in this routine. Therefore, all pointers in the given structure are assumed to be NULL when this function is called. A call to both *LScreateEnv*() and *LScreateModel*(), however, must have been made first to properly establish the structure.
- When reading a formatted MPS file, all text is read literally, including spaces, in the columns of that field. For example, if "ABC DEF" is the text provided in the field for row names in the ROWS section of the MPS file, then this is taken as the row name. If " ABC DEF" (note the initial space) appears as another row name, then this name is treated literally as the text between the quotes and is therefore different from "ABC DEF". MPS file format details are given in Appendix B, *MPS File Format*.
- When reading an unformatted MPS file, the row and column names should not contain spaces. Spaces within the names will tend to generate errors and any leading or trailing spaces will be ignored (making " ABC" equivalent to "ABC"). Note, "unformatted" in the sense used here, does not mean binary format as used by some compilers. The low level file format is still standard ASCII text.
- When the file type is set to LS_FORMATTED_MPS, all names will have 8 characters. When the file type is set to LS_UNFORMATTED_MPS, the length of a name is only restricted by the maximum length of a line, which is 256 characters.
- To minimize the probability of a file open error, it is useful to give the fully specified file path name (e.g., *c:\mydir\myfile.mps*) rather than just *myfile.mps*.

LSreadMPIFile()

Description:

Reads the model in MPI format from the given file and stores the problem data in the given model structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSreadMPIFile(pLSmodel pModel, char *pszFname)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the model. To obtain a pointer to a model structure, see *LScreateModel*().
pszFname	A pointer to a null terminated string containing the path and name of the MPI format file.

Remarks:

- Details for the MPI file format are given in Appendix D, *MPI File Format*.

LSwriteMPIFile()

Description:

Writes the given model in MPI format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSwriteMPIFile(pLSmodel pModel, char *pszFname)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the model.
pszFname	A pointer to a null terminated string containing the path and name of the MPI format file.

Remarks:

- The model must have been loaded via LSloadInstruct call previously.
- Details for the MPI file format are given in Appendix D, *MPI File Format*.

LSreadBasis()

Description:

Reads an initial basis from the given file in the specified format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSreadBasis(pLSmodel pModel, char *pszFname, int nFormat)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the model
pszFname	A pointer to a null terminated string containing the path and name of the basis file.
nFormat	An integer parameter indicating the format of the file to be read. Possible values are • LS_BASFILE_BIN : Binary format (default) • LS_BASFILE_MPS : MPS file format. • LS_BASFILE_TXT : Space delimited text format.

Remarks:

- LS_BASFILE_MPS option requires the variable and constraint names in the resident model and the basis MPS file to match.

LSwriteBasis()

Description:

Writes the resident basis to the given file in the specified format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSwriteBasis(pLSmodel pModel, char *pszFname, int nFormat)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the model
pszFname	A pointer to a null terminated string containing the path and name of the basis file.

nFormat	An integer parameter indicating the format of the file to be written. Possible values are • LS_BASFILE_BIN : Binary format (default) • LS_BASFILE_MPS : MPS file format. • LS_BASFILE_TXT : Space delimited text format.

Remarks:

- LS_BASFILE_MPS option requires the variable and constraint names in the resident model and the basis MPS file to match.

LSwriteDualLINDOFile()

Description:

Writes the dual of a given problem to a file in LINDO format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSwriteDualLINDOFile(pLSmodel pModel, char *pszFname, int nObjsense)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* containing the model to be written to a LINDO format file.
pszFname	A pointer to a null terminated character string containing the path and name of the file to which the dual model should be written.
nObjsense	An integer specifying if the dual problem will be posed as a maximization or minimization problem. The possible values are LS_MAX and LS_MIN.

Remarks:

- The primal model is assumed to be a linear model. Presence of integrality restrictions and quadratic terms will be ignored when writing the dual problem.

LSwriteDualMPSFile()

Description:

Writes the dual of a given problem to a file in MPS format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*.

Prototype:

int	LSwriteDualMPSFile(pLSmodel pModel, char *pszFname, int nFormat, int nObjsense)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* containing the model to be written to a MPS format file.
pszFname	A pointer to a null terminated character string containing the path and name of the file to which the dual model should be written.
nFormat	An integer parameter indicating the format of the file to be written. LS_FORMATTED_MPS indicates the file is to be formatted, while LS_UNFORMATTED_MPS indicates unformatted output.
nObjsense	An integer specifying if the dual problem will be posed as a maximization or minimization problem. The possible values are LS_MAX and LS_MIN.

Remarks:

- The primal model is assumed to be a linear model. Presence of integrality restrictions and quadratic terms in the primal model will be ignored when creating the dual problem.

LSwriteIIS()

Description:

Writes the IIS of an infeasible LP to a file in LINDO file format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*.

Prototype:

int	LSwriteIIS(pLSmodel pModel, char *pszFname)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* containing the infeasible model for which the IIS has already been computed.
pszFname	A character string containing the path and name of the file to which the IIS should be written.

Remarks:

- *LSfindIIS*() can be used to find the IIS of an infeasible LP.

LSwriteIUS()

Description:

Writes the IUS of an unbounded LP to a file in LINDO file format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*.

Prototype:

int	LSwriteIUS(pLSmodel pModel, char *pszFname)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* containing the unbounded model for which the IUS has already been computed.
pszFname	A character string containing the path and name of the file to which the IUS should be written.

Remarks:

- *LSfindIUS*() can be used to find IUS of an unbounded linear model.

LSwriteLINDOFile()

Description:

Writes the given problem to a file in LINDO format. Model must be linear.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*.

Prototype:

int	LSwriteLINDOFile(pLSmodel pModel, char *pszFname)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* containing the model to be written to a LINDO format file.
pszFname	A pointer to a null terminated character string containing the path and name of the file to which the model should be written.

Remarks:

- Details for the LINDO file format are given in Appendix C, *LINDO File Format*.
- To read a model in LINDO format, see *LSreadLINDOFile*().
- To write a model in MPS format, see *LSwriteMPSFile*().

LSwriteLINGOFile()

Description:

Writes the given problem to a file in LINGO format. Model must be linear.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*.

Prototype:

int	LSwriteLINGOFile(pLSmodel pModel, char *pszFname)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* containing the model to be written to a LINGO file.
pszFname	A pointer to a null terminated string containing the path and name of the file to which the model should be written.

Remarks:

- To write a model in LINDO format, see *LSwriteLINDOFile*().
- To write a model in MPS format, see *LSwriteMPSFile*().

LSwriteMPSFile()

Description:

Writes the given problem to a specified file in MPS format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSwriteMPSFile(pLSmodel pModel, char *pszFname, int nFormat)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* containing the model to be written to an MPS file.
pszFname	A pointer to a null terminated string containing the path and name of the file to which the model should be written.
nFormat	An integer parameter indicating the format of the file to be written. LS_FORMATTED_MPS indicates the file is to be formatted, while LS_UNFORMATTED_MPS indicates unformatted output.

Remarks:

- If the name vectors in the model are not set, then the problem title will be "NO_TITLE"; the objective name will be "OBJ"; the column names will be "C0000001", "C0000002", etc.; and the row names will be "R0000001", "R0000002", etc. The name vectors may be set via a call to *LSloadNameData*().
- When using formatted output, this routine writes in the standard MPS format using 8 character names. Longer names are truncated to 8 characters. Therefore, care must be taken when using longer names, since two unique names such as "012345678" and "012345679" will both be treated as "01234567". If your model has names longer than eight characters, you should use unformatted output.
- Details for the MPS file format are given in Appendix B, *MPS File Format*.
- To read a model in MPS format, see *LSreadMPSFile*().
- To write a model in LINDO format, see *LSwriteLINDOFile*().

LSwriteSolution()

Description:

Writes the LP solution to a file.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSwriteSolution(pLSmodel pModel, char *pszFname)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* containing the model to write the LP solution for.
pszFname	A character string containing the path and name of the file to which the solution should be written.

LSreadSMPSFile ()

Description:

This subroutine is the top level input routine. It first reads a core-file in the MPS format. It then calls further subroutines to read time and stoch files whose format are laid out in Appendix E.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSreadSMPSFile (pLSmodel pModel, char * coreFile, char * timeFile, char * stocFile, int nMPStype)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
coreFile	A string specifying the name of the CORE file in MPS format.
timeFile	A string specifying the name of the TIME file.
stocFile	A string specifying the name of the STOCH file.
nMPStype	An integer parameter indicating whether the MPS file is formatted or not. Possible values are: • LS_FORMATTED_MPS • LS_UNFORMATTED_MPS • LS_FORMATTED_MPS_COMP

Remarks:

Refer to appendix for details on SMPS format.

LSreadSMPIFile()

Description:

Read an SP model in SMPI file format in to the given model instance. . It first reads a core-file in the MPI format. It then calls further subroutines to read time and stoch files whose format are laid out in Appendix F.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSreadSMPIFile (pLSmodel pModel, char *coreFile, char *timeFile, char *stocFile)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
coreFile	A string specifying the name of the CORE file in MPS format.
timeFile	A string specifying the name of the TIME file.
stocFile	A string specifying the name of the STOCH file.

Remarks:

Refer to appendix for details on SMPI format.

LSwriteSMPIFile()

Description:

Writes the CORE,TIME,STOCH files for SP models in SMPI format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*.

Prototype:

int	LSwriteSMPIFile (pLSmodel pModel, char * coreFile, char * timeFile, char * stocFile)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
coreFile	A string specifying the name of the CORE file in MPI format.
timeFile	A string specifying the name of the TIME file.
stocFile	A string specifying the name of the STOCH file.

LSwriteSMPSFile ()

Description:

Writes the CORE,TIME,STOCH files for SP models in SMPS format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSwriteSMPSFile (pLSmodel pModel, char * coreFile, char * timeFile, char * stocFile, int nMPStype)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
coreFile	A string specifying the name of the CORE file in MPS format.
timeFile	A string specifying the name of the TIME file.
stocFile	A string specifying the name of the STOCH file.
nMPStype	An integer parameter indicating whether the MPS file is formatted or not. Possible values are: LS_FORMATTED_MPS LS_UNFORMATTED_MPS LS_FORMATTED_MPS_COMP

LSwriteDeteqMPSFile ()

Description:

Writes the deterministic equivalent for the SP model in MPS format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSwriteDeteqMPSFile (pLSmodel pModel, char * mpsFile, int nMPStype, int iDeqType)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
mpsFile	A string specifying the name of the MPS file
nMPStype	An integer parameter indicating whether the MPS file is formatted or not. Possible values are: • LS_FORMATTED_MPS • LS_UNFORMATTED_MPS • LS_FORMATTED_MPS_COMP
iDeqType	An integer specifying the type of the deterministic equivalent. Possible values are • LS_DETEQ_IMPLICIT • LS_DETEQ_EXPLICIT (default).

LSwriteDeteqLINDOFile ()

Description:

Writes the deterministic equivalent (DEQ) of the SP models in LINDO format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSwriteDeteqLINDOFile (pLSmodel pModel, char * ltxFile, int iDeqType)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
ltxFile	An string specifying the name of the LINDO file.
iDeqType	Type of the the deterministic equivalent. Possible values are • LS_DETEQ_IMPLICIT • LS_DETEQ_EXPLICIT (default).

LSgetNodeReducedCost ()

Description:

Returns the reduced cost for the specified node.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetNodeReducedCost (pLSmodel pModel, int iScenario, int iStage, double * padD)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario the node belongs to.
iStage	An integer specifying the stage the node belongs to.
padD	A double array to return specified nodes's dual solution The length of this vector is equal to the number of variables in the stage associated with the node. It is assumed that memory has been allocated for this vector.

Remarks:

The number of variables or constraints in a stage can be accessed via **LSgetStocInfo()**.

LSwriteScenarioSolutionFile ()

Description:

Writes the scenario solution to a file.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSwriteScenarioSolutionFile (pLSmodel pModel, int iScenario, char * szFname)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario to write the solution for.
szFname	A null terminated string containing the file name. If set to NULL, then the results are printed to stdout

LSwriteNodeSolutionFile ()

Description:

Writes the node solution to a file.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSwriteNodeSolutionFile (pLSmodel pModel, int iScenario, int iStage, char * szFname)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario number the node belongs to.
iStage	An integer specifying the stage the node belongs to.
szFname	A null terminated string containing the file name. If set to NULL, then the results are printed to stdout.

LSwriteScenarioMPIFile ()

Description:

Write scenario model in MPI format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSwriteNodeSolutionFile (pLSmodel pModel, int iScenario, int iStage, char * szFname)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario to write in MPI format..
scenFile	A null terminated string specifying file to write the scenario model..

LSwriteScenarioMPSFile ()

Description:

Write a specific scenario model in MPS format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSwriteScenarioMPSFile (pLSmodel pModel, int iScenario, char * scenFile, int nMPStype)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario to write in MPS format..
scenFile	A null terminated string specifying file to write the scenario model.
nMPStype	An integer parameter indicating whether the MPS file is formatted or not. Possible values are: • LS_FORMATTED_MPS • LS_UNFORMATTED_MPS • LS_FORMATTED_MPS_COMP

LSwriteScenarioLINDOFile ()

Description:

Write scenario model in LINDO format.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSwriteScenarioLINDOFile (pLSmodel pModel, int iScenario, char * scenFile)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario to write in MPI format..
scenFile	A null terminated string specifying file to write the scenario model.

Parameter Setting and Retrieving Routines

The routines in this section allow you to set and retrieve system parameter values. Each of these routines is distinguished on three dimensions:

1. The parameter being dealt with is void, double precision, or integer.
2. The routine either gets or sets the specified parameter's value.
3. The parameter being dealt with is in either a model space or an environment space.

The various permutations of these three options result in a total of fifteen routines. A brief listing of these routines and their usage is listed in the following table:

Routine	Parameter Type	Action	Location
LSgetEnvParameter()	Void	Gets	Environment
LSgetEnvDouParameter()	Double	Gets	Environment
LSgetEnvIntParameter()	Integer	Gets	Environment
LSgetModelParameter()	Void	Gets	Model
LSgetModelDouParameter()	Double	Gets	Model
LSgetModelIntParameter()	Integer	Gets	Model
LSsetEnvParameter()	Void	Sets	Environment
LSsetEnvDouParameter()	Double	Sets	Environment
LSsetEnvIntParameter()	Integer	Sets	Environment
LSsetModelParameter()	Void	Sets	Model
LSsetModelDouParameter()	Double	Sets	Model
LSsetModelIntParameter()	Integer	Sets	Model
LSreadEnvParameter()	N/A	Reads	Environment
LSwriteEnvParameter()	N/A	Writes	Environment
LSreadModelParameter()	N/A	Reads	Model
LSwriteModelParameter()	N/A	Writes	Model

These fifteen functions are documented in detail immediately below. The list of parameters that may be referenced through these routines is given in the section *Available Parameters*. This lists, each of the parameter's data type (integer or double) and whether they are available as part of the environment or model. The parameters available to be set for the environment are also available to be set for the model. However, some of the parameters available to be set for the model are not available to be set for the environment.

All parameters are assigned default (initial) values during environment and model creation. These defaults work best for general purpose. However, there may be cases where users prefer to work with different settings for a subset of the available parameters. When a model is created, it inherits the parameter values in the environment it belongs to. Changes to the parameter values in the model do not affect the parameter values currently set in the environment. Similarly, once a model is created in an

environment, subsequent changes in the environment parameters do not affect the parameter settings in the model. During the optimization process, the solver uses the parameter settings in the model space. If a parameter is not part of the model space, then the solver uses the value in the environment space.

LSgetEnvParameter()

Description:

Retrieves a parameter for a specified environment.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetEnvParameter(pLSenv pEnv, int nParameter, void *pvValue)

Input Arguments:

Name	Description
pEnv	Pointer to an instance of *LSenv*.
nParameter	An integer macro (e.g., LS_IPARAM_STATUS).

Output Arguments:

Name	Description
pvValue	On return, *pvValue will contain the parameter's value. The user is responsible for allocating sufficient memory to store the parameter value.

Remarks:

- The available parameters are described in the *Available Parameters* section below.
- For retrieving model parameters, use *LSgetModelParameter*().

LSgetEnvDouParameter()

Description:

Retrieves a double precision parameter for a specified environment.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetEnvDouParameter(pLSenv pEnv, int nParameter, double *pdVal)

Input Arguments:

Name	Description
pEnv	Pointer to an instance of *LSenv*.
nParameter	An integer macro referring to a double precision parameter (e.g., LS_DPARAM_SOLVER_FEASTOL).

Output Arguments:

Name	Description
pdVal	A pointer to a double precision variable. On return, *pdVal will contain the parameter's value.

Remarks:

- The available parameters are described in the *Available Parameters* section below.
- For retrieving double precision model parameters, use *LSgetModelDouParameter*().
- For retrieving integer environment parameters, use *LSgetEnvIntParameter*().

LSgetEnvIntParameter()

Description:

Retrieves an integer parameter for a specified environment.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetEnvIntParameter(pLSenv pEnv, int nParameter, int *pnVal)

Input Arguments:

Name	Description
pEnv	Pointer to an instance of *LSenv*.
nParameter	An integer macro referring to an integer parameter (e.g., LS_IPARAM_LP_ITRLMT).

Output Arguments:

Name	Description
pnVal	A pointer to an integer variable. On return, *pnVal will contain the parameter's value.

Remarks:

- The available parameters are described in the *Available Parameters* section below.
- For retrieving integer model parameters, use *LSgetModelIntParameter*().
- For retrieving double precision environment parameters, use *LSgetEnvDouParameter*().

LSgetModelParameter()

Description:

Retrieves a parameter or status variable for a specified model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetModelParameter(pLSmodel pModel, int nParameter, void *pvValue)

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.
nParameter	An integer macro (e.g., LS_IPARAM_STATUS).

Output Arguments:

Name	Description
pvValue	On return, *pvValue will contain the parameter's value. The user is responsible for allocating sufficient memory to store the parameter value.

Remarks:

- The available parameters are described in the *Available Parameters* section below.
- For retrieving environment parameters, use *LSgetEnvParameter*().

LSgetModelDouParameter

Description:

Retrieves a double precision parameter for a specified model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetModelDouParameter(pLSmodel pModel, int nParameter, double *pdVal)

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.
nParameter	An integer macro referring to a double precision parameter (e.g., LS_DPARAM_MIP_RELOPTTOL).

Output Arguments:

Name	Description
pdVal	A pointer to a double precision variable. On return, *pdVal will contain the parameter's value.

Remarks:

- The available parameters are described in the *Available Parameters* section below.
- For retrieving double precision environment parameters, use *LSgetEnvDouParameter*().
- For retrieving integer model parameters, use *LSgetModelIntParameter*().

LSgetModelIntParameter()

Description:

Retrieves an integer parameter for a specified model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetModelIntParameter(pLSmodel pModel, int nParameter, int *pnVal)

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.
nParameter	An integer macro referring to an integer parameter (e.g., LS_IPARAM_LP_ITRLMT).

Output Arguments:

Name	Description
pnVal	A pointer to an integer variable. On return, *pnVal will contain the parameter's value.

Remarks:

- The available parameters are described in the *Available Parameters* section below.
- For retrieving integer environment parameters, use *LSgetEnvIntParameter*().
- For retrieving double precision model parameters, use *LSgetModelDouParameter*().

LSsetEnvParameter()

Description:

Sets a parameter for a specified environment.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSsetEnvParameter(pLSenv pEnv, int nParameter, void *pvValue)

Input Arguments:

Name	Description
pEnv	Pointer to an instance of *LSenv*.
nParameter	An integer macro (e.g., LS_DPARAM_SOLVER_FEASTOL).
pvValue	A variable containing the parameter's new value.

Remarks:

- The available parameters are described in the *Available Parameters* section below.
- For setting model parameters, use *LSsetModelParameter*().

LSsetEnvDouParameter()

Description:

Sets a double precision parameter for a specified environment.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*.

Prototype:

int	LSsetEnvDouParameter(pLSenv pEnv, int nParameter, double dVal)

Input Arguments:

Name	Description
pEnv	Pointer to an instance of *LSenv*.
nParameter	An integer macro referring to a double precision parameter (e.g., LS_DPARAM_SOLVER_FEASTOL).
dVal	A double precision variable containing the parameter's new value.

Remarks:

- The available parameters are described in the *Available Parameters* section below.
- For setting integer environment parameters, use *LSsetEnvIntParameter*().
- For setting double precision model parameters, use *LSsetModelDouParameter*().

LSsetEnvIntParameter()

Description:

Sets an integer parameter for a specified environment.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSsetEnvIntParameter(pLSenv pEnv, int nParameter, int nVal)

Input Arguments:

Name	Description
pEnv	Pointer to an instance of *LSenv*.
nParameter	An integer macro referring to an integer parameter (e.g., LS_IPARAM_LP_PRELEVEL).
nVal	An integer variable containing the parameter's new value.

Remarks:

- The available parameters are described in *Available Parameters* below.
- For setting double precision environment parameters, use *LSsetEnvDouParameter*().
- For setting integer model parameters, use *LSsetModelIntParameter*().

LSsetModelParameter()

Description:

Sets a parameter for a specified model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSsetModelParameter(pLSmodel pModel, int nParameter, void *pvValue)

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.
nParameter	An integer macro (e.g., LS_IPARAM_LP_ITRLMT).
PvValue	A variable containing the parameter's new value.

Remarks:

- The available parameters are described in *Available Parameters* below.
- For setting environment parameters, use *LSsetEnvParameter*().

LSsetModelDouParameter()

Description:

Sets a double precision parameter for a specified model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*.

Prototype:

int	LSsetModelDouParameter(pLSmodel pModel, int nParameter, double dVal)

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.
nParameter	An integer macro referring to a double precision parameter (e.g., LS_DPARAM_SOLVER_FEASTOL).
dVal	A double precision variable containing the parameter's new value.

Remarks:

- The available parameters are described in the *Available Parameters* section below.
- For setting integer model parameters, use *LSsetModelIntParameter*().
- For setting double precision environment parameters, use *LSsetEnvDouParameter*().

LSsetModelIntParameter()

Description:

Sets an integer parameter for a specified environment.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*.

Prototype:

int	LSsetModelIntParameter(pLSmodel pModel, int nParameter, int nVal)

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.
nParameter	An integer macro referring to an integer parameter (e.g., LS_IPARAM_TIMLIM).
nVal	An integer variable containing the parameter's new value.

Remarks:

- The available parameters are described in the *Available Parameters* section below.
- For setting double precision model parameters, use *LSsetModelDouParameter*().
- For setting integer environment parameters, use LSsetEnvIntParameter().

LSreadEnvParameter()

Description:

Reads environment parameters from a parameter file.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSreadEnvParameter(pLSenv pEnv, char *pszFname)

Input Arguments:

Name	Description
pEnv	Pointer to an instance of *LSenv*.
pszFname	A null-terminated string containing the path and name of the file from which parameters will be read.

Remarks:

- The available parameters are described in the *Available Parameters* section below.
- For retrieving environment parameters, use *LSgetModelParameter*().
- For an example parameter file, see *lindo.par* in the distribution.

LSreadModelParameter()

Description:

Reads model parameters from a parameter file.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSreadModelParameter(pLSmodel pModel, char *pszFname)

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.
pszFname	A null-terminated string containing the path and name of the file from which parameters will be read.

Remarks:

- The available parameters are described in the *Available Parameters* section below.
- For retrieving environment parameters, use *LSgetEnvParameter*().

LSwriteEnvParameter()

Description:

Writes environment parameters to a parameter file.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSwriteEnvParameter(pLSenv pEnv, char *pszFname)

Input Arguments:

Name	Description
pEnv	Pointer to an instance of *LSenv*.
pszFname	A null-terminated string containing the path and name of the file to which parameters will be written.

Remarks:

- LSmodel objects inherit default parameter values from the LSenv object they belong.

LSwriteModelParameter()

Description:

Writes model parameters to a parameter file.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSwriteModelParameter(pLSmodel pModel, char *pszFname)

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.
pszFname	A null-terminated string containing the path and name of the file to which parameters will be written.

Remarks:

- LSmodel objects inherit default parameter values from the LSenv object they belong.

Available Parameters

In this section, a detailed description of all the parameters available through the *LSgetxxxyyyParameter*() and *LSsetxxxyyyParameter*() routines are described. These parameters are defined in the *lindo.h* header file in the LSparameter enumerated type definition. The parameters that start with LS_IPARAM corresponds to *integer* type parameters. Similarly, parameters that start with LS_DPARAM correspond to *double* type parameters.

Note: For details on the relationship between environment and model parameters, see the *Parameter Setting and Retrieving Routines* section above.

General Solver Parameters

Name	Available for	Description
LS_IPARAM_CHECK_FOR_ERRORS	Environment, Model	This is a flag indicating if the loaded model will be checked for errors. Possible values are 0 and 1. 1 means that the loaded model will be checked for errors. 0 means it will not. The default is 0.
LS_IPARAM_SPLEX_REFACFRQ	Environment, Model	This is a positive integer scalar referring to the simplex iterations between two consecutive basis re-factorizations. For numerically unstable models, setting this parameter to smaller values may help. Range for possible values is (0,inf). The default is 100.
LS_IPARAM_BARRIER_SOLVER	Environment, Model	This is the type of barrier method to be used for solving the referred model. This macro is reserved for future use.
LS_IPARAM_ALLOW_CNTRLBREAK	Environment	This flag controls if the user can interrupt the solver using the CTRL+C keys. The default is 1 (on).
LS_IPARAM_SOL_REPORT_STYLE	Model	This controls the solution report style produced. Possible values are 0 (default) and 1. The latter produces solution reports in LINDO API 1.x style.
LS_DPARAM_CALLBACKFREQ	Environment, Model	This controls the frequency with which the solver calls back to your optionally supplied callback routine. Range for possible values is [0,inf). The default value for this option is 0.5, meaning the solver calls back once every ½ second.
LS_IPARAM_INSTRUCT_LOADTYPE	Environment, Model	This is reserved for internal use only.

LS_DPARAM_SOLVER_CUTOFFVAL	Model	If the optimal objective value of the LP being solved is shown to be worse than this (e.g., if the dual simplex method is being used), then the solver will exit without finding a feasible solution. This is a way of saving computer time if there is no sufficiently attractive solution. Range for possible values is (-inf,inf).
LS_IPARAM_MPS_OBJ_WRITESTYLE	Model	Standard MPS format assumes that the underlying model is of minimization type. This flag indicates how to handle 'maximization' type models when exporting in MPS format. Possible values are: • LS_MPS_USE_MAX_FLIP: Export the model as a minimization problem after flipping the sign of the objective. This is the default. • LS_MPS_USE_MAX_NOTE: Export the model as minimization type without flipping the objective function but a comment is printed in the file that the model is of maximization type. • LS_MPS_USE_MAX_CARD: Export the model using the non-standard 'MAX' operator. Some MPS parsers, including LINDO API can process MAX operator.
LS_IPARAM_BARRIER_PROB_TO_SOLVE	Environment, Model	This is the orientation of the problem to be solved by the barrier solver. The possible values are: • LS_PROB_SOLVE_FREE: solver decides. This is the default. • LS_PROB_SOLVE_PRIMAL: solve the primal model, • LS_PROB_SOLVE_DUAL: solve the dual model.

LS_IPARAM_DECOMPOSITION_TYPE	Model	This refers to the type of decomposition to be performed on a linear or mixed integer model. The possible values are identified with the following macros: • LS_LINK_BLOCKS_FREE: The solver decides which type of decomposition to use. • LS_LINK_BLOCKS_SELF: The solver does not perform any decompositions and uses the original model. This is the default. • LS_LINK_BLOCKS_NONE: Attempt total decomposition (no linking rows or columns). • LS_LINK_BLOCKS_COLS: The decomposed model will have dual angular structure (linking columns). • LS_LINK_BLOCKS_ROWS: The decomposed model will have block angular structure (linking rows). • LS_LINK_BLOCKS_BOTH: The decomposed model will have both dual and block angular structure (linking rows and columns). For more information on decomposing models, refer to Chapter 10, *Analyzing Models and Solutions*.
LS_DPARAM_SOLVER_FEASTOL	Environment, Model	This is the feasibility tolerance. A constraint is considered violated if the artificial, slack, or surplus variable associated with the constraint violates its lower or upper bounds by the feasibility tolerance. Range for possible values is [1e-16,inf). The default value is 1.0e-7.
LS_DPARAM_SOLVER_OPTTOL	Environment, Model	This is the optimality tolerance. It is also referred to as the dual feasibility tolerance. A dual slack (reduced cost) is considered violated if it violates its lower bound by the optimality tolerance. Range for possible values is [1e-16,inf). The default value is 1.0e-7.

LS_IPARAM_LP_SCALE	Environment, Model	This is the scaling flag for linear models's, both Simplex methods as well as barrier and MILP. Scaling multiplies the rows and columns of the model by appropriate factors so as to reduce the range of coefficient This tends to reduce numerical difficulties. A value of 1 means scaling is performed. A 0 means scaling is suppressed. The default is 1.
LS_IPARAM_LP_ITRLMT	Environment, Model	This is a limit on the number of iterations the solver will perform before terminating. If this value is a nonnegative integer, then it will be used as an upper bound on the number of iterations the solver will perform. If this value is -1, then no iteration limit will be used. The solution may be infeasible. Range for possible values is [-1,INT_MAX). The default is -1.
LS_DPARAM_LP_ITRLMT	Environment, Model	This is a limit on the number of iterations (stored as a double) the solver will perform before terminating. If this value is a nonnegative double, then it will be used as an upper bound on the number of iterations the solver will perform. If this value is -1.0, then no iteration limit will be used. The solution may be infeasible. Range for possible values is [-1.0,inf). The default is -1.0.
LS_IPARAM_SOLVER_IUSOL	Environment, Model	This is a flag that, when set to 1, will force the solver to compute a basic solution to an infeasible model that minimizes the sum of infeasibilities and a basic feasible solution to an unbounded problem from which an extreme direction originates. When set to 0, the solver will return with an appropriate status flag as soon as infeasibility or unboundedness is detected. If infeasibility or unboundedness is declared with presolver's determination, no solution will be computed. The default is 0.

LS_IPARAM_LP_PRINTLEVEL	Environment, Model	This controls the level of trace output printed by the simplex and barrier solvers. 0 means no trace output. Higher values lead to more trace output. Range for possible values is [0,inf). The default is 0.
LS_DPARAM_OBJPRINTMUL	Model	When printing the objective value, it will first be multiplied by the value of this parameter. For example, you may wish to set it to -1.0 if the original problem was a maximization problem, but it is being solved as a minimization problem. Range for possible values is (-inf,inf). The default value is 1.0.
LS_IPARAM_OBJSENSE	Model	Use this parameter to set the sense of the objective function. The default value is LS_MIN, for minimization. Set this parameter to LS_MAX if you want to maximize the objective.
LS_IPARAM_SPLEX_PPRICING	Environment, Model	This is the pricing option to be used by the primal simplex method. Possible values are: -1: Solver decides the primal pricing method (default). 0: Partial pricing. 1: Devex
LS_IPARAM_SPLEX_DPRICING	Environment, Model	This is the pricing option to be used by the dual simplex method. Possible values are: -1: Solver decides. 0: Dantzig's rule (partial pricing). 1: Dantzig's rule (full pricing with fallback to partial). 2: Steepest edge rule. 3: Dual Devex rule. 4: Approximate dual Devex rule.
LS_IPARAM_SOLVER_RESTART	Environment, Model	This is the starting basis flag. Possible values are 1 or 0. 1 means LINDO API will start from a cold basis discarding any basis resident in memory. 0 means LINDO API will perform warm starts using any basis currently in memory. The default is 0.

LS_IPARAM_PROB_TO_SOLVE	Environment, Model	This controls whether the explicit primal or dual form of the given LP problem will be solved. Possible values are: 0: Solver decides (default). 1: Explicit primal form. 2: Explicit dual form.
LS_IPARAM_SOLVER_IPMSOL	Environment, Model	This flag controls whether a basis crossover will be performed when solving LPs with the barrier solver. A value of 0 indicates that a crossover to a basic solution will be performed. If the value is 1, then the barrier solution will be left intact. For example, if alternate optima exist, the barrier method will return a solution that is, loosely speaking, the average of all alternate optima. The default is 0.
LS_IPARAM_SOLVER_TIMLMT	Model	This is a time limit in seconds for the LP solver. The default value of -1 imposes no time limit. If LS_IPARAM_TIMLMT < -1, then an error is returned. Range for possible values is [-1, inf).
LS_IPARAM_SOLVER_USECUTOFFVAL	Environment, Model	This is a flag for the parameter LS_DPARAM_SOLVER_CUTOFFVAL. The possible value of 0 means LS_DPARAM_SOLVER_CUTOFFVAL is not used, else it is used as defined. Possible values are 0 (no), 1 (yes) and -1 (the solver decides). The default is 0.
LS_IPARAM_VER_NUMBER	Environment	This is the version number of LINDO API. This value cannot be set.

LS_IPARAM_LP_PRELEVEL	Environment, Model	This controls the amount and type of LP pre-solving to be used. Possible values in bit-mask form are: Simple pre-solving +2 Primal based +4 Coefficient reduction +8 Elimination +16 Dual column based +32 Dual row based +64 Use Max pass limit +128 The default value is: 254 = 2+4+8+16+32+64+128.
LS_IPARAM_SOLVER_PRE_ELIM_FILL	Environment, Model	This is a nonnegative value that controls the fill-in introduced by the eliminations during pre-solve. Smaller values could help when the total nonzeros in the presolved model is significantly more than the original model. Range for possible values is [0,inf). The default is 1000.
LS_IPARAM_SPLEX_DUAL_PHASE	Environment, Model	This controls the dual simplex strategy, single-phase versus two-phase. The possible values are 0,1 and 2. The default is 0, i.e. the solver decides.

Nonlinear Optimization Parameters

LS_IPARAM_NLP_SOLVE_AS_LP	Environment, Model	This is a flag indicating if the nonlinear model will be solved as an LP. Possible values are 0 and 1. 1 means that an LP using first order approximations of the nonlinear terms in the model will be used when optimizing the model with the *LSoptimize*() function. The default is 0.
LS_IPARAM_NLP_SOLVER	Environment, Model	This refers to the type of nonlinear solver. The possible values are: • LS_NMETHOD_FREE: solver decides, • LS_NMETHOD_CONOPT: uses CONOPT's reduced gradient solver. This is the default. • LS_NMETHOD_MSW_GRG: uses CONOPT with multistart feature enabled.

LS_IPARAM_NLP_SUBSOLVER	Environment, Model	This controls the type of linear solver to be used for solving linear sub problems when solving nonlinear models. The possible values are: • LS_METHOD_PSIMPLEX: primal simplex method. This is the default. • LS_METHOD_DSIMPLEX: dual simplex method, • LS_METHOD_BARRIER: barrier solver with or without crossover.
LS_DPARAM_NLP_PSTEP_FINITEDIFF	Environment, Model	This controls the value of the step length in computing the derivatives using finite differences. Range for possible values is (0, inf). The default value is 5.0E-07.
LS_IPARAM_NLP_USE_CRASH	Environment, Model	This is a flag indicating if an initial solution will be computed using simple crash routines. Possible values are 0 (no), 1 (yes) and -1 (the solver decides). The default is 0.
LS_IPARAM_NLP_USE_STEEPEDGE	Environment, Model	This is a flag indicating if steepest edge directions should be used in updating the solution. Possible values are 0 (no), 1 (yes) and -1 (the solver decides). The default value is 0.
LS_IPARAM_NLP_USE_SLP	Environment, Model	This is a flag indicating if sequential linear programming step directions should be used in updating the solution. Possible values are 0 (no), 1 (yes) and -1 (the solver decides). The default value is 0.
LS_IPARAM_NLP_USE_SELCONEVAL	Environment, Model	This is a flag indicating if selective constraint evaluations will be performed in solving a nonlinear model. Possible values are 0 (no), 1 (yes) and -1 (the solver decides). The default value is 0.

LS_IPARAM_NLP_PRELEVEL	Environment, Model	This controls the amount and type of NLP pre-solving. Possible options are: Simple pre-solving +2 Probing +4 Coefficient reduction +8 Elimination +16 Dual reductions +32 Use dual information +64 Maximum pass +512 The default value is: 126 = 2+4+8+16+32+64.
LS_IPARAM_NLP_AUTODERIV	Environment, Model	This is a flag to indicate if automatic differentiation is the method of choice for computing derivatives and select the type of differentiation. If the value is 0, then the Finite Differences approach will be used. If the value is 1, then the forward type of Automatic Differentiation will be used. If the value is 2, then the backward type of Automatic Differentiation will be used. The default is 0. ***Note:*** Automatic Differentiation can be used only with Instruction style input. It is only useful when the instructions are loaded.
LS_IPARAM_NLP_LINEARZ	Environment, Model	This determines the extent to which the solver will attempt to linearize nonlinear models. The available options are 0: Solver decides. 1: No linearization occurs. 2: Linearize *ABS*, *MAX*, and *MIN* functions. 3: Same as option 2 plus *IF*, *AND*, *OR*, *NOT*, and all logical operators (i.e., ≤, = , ≥, and <>) are linearized. The default is 0.

LS_IPARAM_NLP_LINEARITY	Environment, Model	This is used to check the linearity characteristic of the solved model. If the returned value equals 1, then the model is linear or has been completely linearized in the linearization step. Thus, the global optimality of the solution can be ensured. This parameter cannot be set. This parameter will be deprecated in future versions.
LS_IPARAM_NLP_PRINTLEVEL	Environment, Model	This controls the level of trace output printed by the nonlinear solver. 1 means normal trace output. Higher values for this parameter lead to more trace output. Range for possible values is [0,inf). The default is 1.
LS_IPARAM_NLP_FEASCHK	Environment, Model	This input parameter specifies how the NLP solver reports the results when an optimal or local-optimal solution satisfies the feasibililty tolerance (LS_DPARAM_NLP_FEASTOL) of the scaled model but not the original (descaled) one. Possible values for *LS_IPARAM_NLP_FEASCHK* are 0 - Perform no action, accept the final solution (status). 1 - Declare the model status as LS_STATUS_FEASIBLE if maximum violation in the unscaled model is not higher than 10 times of the current feasibililty tolerance (LS_DPARAM_NLP_FEASTOL), otherwise declare the status as LS_STATUS_UNKNOWN. 2 - Declare the model status as LS_STATUS_UNKNOWN if maximum violation in the unscaled model is higher than the current feasibililty tolerance (LS_DPARAM_NLP_FEASTOL). The default is (1).

LS_DPARAM_NLP_FEASTOL	Environment, Model	This is the feasibility tolerance for nonlinear constraints. A constraint is considered violated if the artificial, slack, or surplus variable associated with the constraint violates its lower or upper bounds by the feasibility tolerance. Range for possible values is (0,1). The default value is 1.0e-6.
LS_DPARAM_NLP_REDGTOL	Environment, Model	This is the tolerance for the gradients of nonlinear functions. The (projected) gradient of a function is considered to be the zero-vector if its norm is below this tolerance. Range for possible values is (0,1). The default value is 1.0e-7.
LS_IPARAM_NLP_DERIV_DIFFTYPE	Environment, Model	This is a flag indicating the technique used in computing derivatives with *Finite Differences*. The possible values are: • LS_DERIV_FREE: the solver decides (default), • LS_DERIV_FORWARD_DIFFERENCE: use forward differencing method, • LS_DERIV_BACKWARD_DIFFERENCE: use backward differencing method, • LS_DERIV_CENTER_DIFFERENCE: use center differencing method.
LS_IPARAM_NLP_ITRLMT	Environment, Model	This controls the iteration limit on the number of nonlinear iterations performed. Range for possible values is [0,INT_MAX). The default is 20000.
LS_IPARAM_NLP_STARTPOINT	Environment, Model	This is a flag indicating if the nonlinear solver should accept initial starting solutions. Possible values are 0 (no), 1 (yes). The default is 1.
LS_IPARAM_NLP_CONVEXRELAX	Environment, Model	This is reserved for internal use only.
LS_IPARAM_NLP_CR_ALG_REFORM	Environment, Model	This is reserved for internal use only.

LS_IPARAM_NLP_QUADCHK	Environment, Model	This is a flag indicating if the nonlinear model should be examined to check if it is a quadratic model. . Possible values are 0 (no), 1 (yes). The default value is 0.
LS_IPARAM_NLP_MAXLOCALSEARCH	Environment, Model	This controls the maximum number of local searches (multistarts) when solving a NLP using the multistart solver. Range for possible values is [0,inf). The default value is 5.
LS_IPARAM_NLP_CONVEX	Environment, Model	This is a flag indicating if the quadratic model is convex or not. If the value is 1, the minimization (maximization) model is convex (concave). This value cannot be set.
LS_IPARAM_NLP_CONOPT_VER	Environment, Model	This specifies the CONOPT version to be used in NLP optimizations. Possible values are 2 and 3 (default).
LS_IPARAM_NLP_USE_LINDO_CRASH	Model	This is a flag indicating if an initial solution will be computed using advanced crash routines. Possible values are 0 (no), 1 (yes) and -1 (the solver decides). The default is 1.
LS_IPARAM_NLP_STALL_ITRLMT	Environment, Model	This specifies the iteration limit before a sequence of non-improving NLP iterations is declared as stalling, thus causing the solver to terminate. The default is 100.
LS_IPARAM_NLP_AUTOHESS	Environment, Model	This is a flag to indicate if Second Order Automatic Differentiation will be performed in solving a nonlinear model. The second order derivatives provide an exact/precise Hessian matrix to the SQP algorithm, which may lead to less iterations and better solutions, but may also be quite expensive in computing time for some cases. If the value is 1, then the Second Order Automatic Differentiation will be used. The default is 0. ***Note:*** Automatic Differentiation can be used only with Instruction style input. It is only useful when the instructions are loaded.

LS_DPARAM_NLP_ITRLMT	Environment, Model	This controls the iteration limit (stored as a double) on the number of nonlinear iterations performed. Range for possible values is [0,inf). The default is 20000.

Interior-Point (Barrier) Solver Parameters

LS_DPARAM_IPM_TOL_INFEAS	Environment, Model	This is the tolerance to declare the model primal or dual infeasible using the interior-point optimizer. A smaller number means the optimizer gets more conservative about declaring the model infeasible. Range for possible values is (0,inf). The default is 1e-08.
LS_DPARAM_IPM_TOL_PATH	Environment, Model	Controls how close the interior-point optimizer follows the central path. A large value of this parameter means the central path is followed very closely. For numerically unstable problems it might help to increase this parameter. Range for possible values is (0,0.5). The default is 1e-08.
LS_DPARAM_IPM_TOL_PFEAS	Environment, Model	Primal feasibility tolerance used for linear and quadratic optimization problems. Range for possible values is (0,inf). The default is 1e-8.
LS_DPARAM_IPM_TOL_REL_STEP	Environment, Model	Relative step size to the boundary for linear and quadratic optimization problems. Range for possible values is (0,9.99999e-1). The default is 9.999e-1.
LS_DPARAM_IPM_TOL_PSAFE	Environment, Model	Controls the initial primal starting point used by the interior-point optimizer. If the interior-point optimizer converges slowly and/or the constraint or variable bounds are very large, then it might be worthwhile to increase this value. Range for possible values is [1e-2,inf). The default is 1.0.
LS_DPARAM_IPM_TOL_DFEAS	Environment, Model	Dual feasibility tolerance used for linear and quadratic optimization problems. Range for possible values is (0,inf). The default is 1e-8.

LS_DPARAM_IPM_TOL_DSAFE	Environment, Model	Controls the initial dual starting point used by the interior-point optimizer. If the interior-point optimizer converges slowly and/or the dual variables associated with constraint or variable bounds are very large, then it might be worthwhile to increase this value. Range for possible values is [1e-4,inf). The default is 1.0.
LS_DPARAM_IPM_TOL_MU_RED	Environment, Model	Relative complementarity gap tolerance. Range for possible values is (0,inf). The default is 1e-16.
LS_DPARAM_IPM_BASIS_REL_TOL_S	Environment, Model	Maximum relative dual bound violation allowed in an optimal basic solution. Range for possible values is (0,inf). The default is 1e-12.
LS_DPARAM_IPM_BASIS_TOL_S	Environment, Model	Maximum absolute dual bound violation in an optimal basic solution. Range for possible values is (0,inf). The default is 1e-07.
LS_DPARAM_IPM_BASIS_TOL_X	Environment, Model	Maximum absolute primal bound violation allowed in an optimal basic solution. Range for possible values is (0,inf). The default is 1e-07.
LS_DPARAM_IPM_BI_LU_TOL_REL_PIV	Environment, Model	Relative pivot tolerance used in the LU factorization in the basis identification procedure. Range for possible values is (0,9.99999e-1). 1e-02.
LS_IPARAM_IPM_MAX_ITERATIONS	Environment, Model	Controls the maximum number of iterations allowed in the interior-point optimizer. Range for possible values is [0,inf). The default is 1000.
LS_IPARAM_IPM_OFF_COL_TRH	Environment, Model	Controls the extent for detecting the offending columns in the Jacobian of the constraint matrix. Range for possible values is [0,inf). 0 means no offending columns will be detected. 1 means offending columns will be detected. In general, increasing the parameter value beyond the default value of 40 does not improve the result.

LS_IPARAM_IPM_NUM_THREADS	Environment, Model	Number of threads to run the interior-point optimizer on. This value should be less than or equal to the actual number of processors on the system. Range for possible values is [0,inf). The default is 1.
LS_IPARAM_SOLVER_CONCURRENT_OPTMODE	Environment, Model	Controls if simplex and interior-point optimizers will run concurrently, 0 means no concurrent runs will be performed,1 means both optimizers will run concurrently if at least two threads exist in system, 2 means both optimizers will run concurrently. The default is 0.

Mixed-Integer Optimization Parameters

LS_IPARAM_MIP_USE_INT_ZERO_TOL	Environment, Model	This flag controls if all MIP calculations would be based on the integrality tolerance specified by LS_DPARAM_MIP_INTTOL. The flag will be disregarded if the following conditions fail to hold • All coefficients of the coefficient matrix and the right-hand side vector are integers • Any continuous variable that is not yet proved to be an implied integer has coefficients all -1 or +1. • All continuous variables have integer bounds or, –inf or +inf • All continuous variables have only one nonzero in each constraint. Possible values are 0 (no), 1 (yes) and -1 (the solver decides). The default for this flag is 0.
LS_IPARAM_MIP_USE_CUTS_HEU	Environment, Model	This flag controls if cut generation is enabled during MIP heuristics. Possible values are 0 (no), 1 (yes) and -1 (the solver decides). The default is -1.
LS_DPARAM_MIP_BIGM_FOR_INTTOL	Environment, Model	This value specifies the threshold for which the coefficient of a binary variable would be considered as big-M (when applicable). Range for possible values is (0,inf). The default is 1.0e8.

LS_IPARAM_MIP_STRONGBRANCHDONUM	Environment, Model	This value specifies the minimum number of variables, among all the candidates, to try the strong branching on. Range for possible values is [0,inf). The default is 3.
LS_IPARAM_MIP_MAKECUT_INACTIVE_COUNT	Environment, Model	This value specifies the threshold for the times a cut could remain active after successive reoptimization during branch-and-bound. If the count is larger than the specified level the solver will inactive the cut. Range for possible values is [0,inf). The default is 20
LS_IPARAM_MIP_PRE_ELIM_FILL	Environment, Model	This is a nonnegative value that controls the fill-in introduced by the eliminations during pre-solveSmaller values could help when the total nonzeros in the presolved model is significantly more than the original model. Range for possible values is [0,inf). The default is 100.
LS_IPARAM_MIP_HEU_MODE	Environment, Model	This controls the MIP heuristic mode. Possible values are: ≤0 solver is free to decide when to stop the heuristic (default), ≤1 solver uses a pre-specified time limit to stop the heuristic. ≤2 solver uses a pre-specified iteration limit to stop the heuristic.
LS_IPARAM_MIP_FP_MODE	Environment, Model	Controls the mode for the feasibility pump heuristic. Possible values are: 0=Off,1=Solver decides,2=On until the first solution, 3=Try to get more than one solutions. The default is 1.
LS_DPARAM_MIP_FP_WEIGTH	Environment, Model	Controls the weight of the objective function in the feasibility pump. Possible values are in the closed interval [0,1]. The default is 1.0.

LS_IPARAM_MIP_FP_OPT_METHOD	Environment, Model	This specifies optimization and reoptimization method for feasibility pump heuristic. Possible values are: 0: Solver decides (default). 1: Use primal method. 2: Use dual simplex. 3: Use barrier solver (with or without basis crossover, depending on LS_IPARAM_SOLVER_IPMSOL parameter setting described above)
LS_DPARAM_MIP_FP_TIMLIM	Environment, Model	This is the time limit in seconds for feasibility pump heuristic. A value of -1 implies no time limit is imposed. Range for possible values is [-1,inf). The default value is 1800.
LS_IPARAM_MIP_FP_ITRLIM	Environment, Model	This is the iteration limit in seconds for feasibility pump heuristic. A value of -1 means no iteration limit is imposed. Range for possible values is [-1,inf). The default value is 500.
LS_IPARAM_MIP_CUTLEVEL_TOP	Environment, Model	This controls the combination of cut types to try at the root node when solving a MIP. Bit settings are used to enable the various cut types. Add the following values to enable the specified cuts: GUB cover +2 Flow cover +4 Lifting +8 Plant location +16 Disaggregation +32 Knapsack cover +64 Lattice +128 Gomory +256 Coefficient reduction +512 GCD +1024 Obj integrality +2048 Basis Cuts +4096 Cardinality Cuts +8192 Disjunctive Cuts +16386 The default is 22526, which means all cut types except objective integrality and cardinality cuts are generated.

LS_IPARAM_MIP_CUTLEVEL_TREE	Environment, Model	This controls the combination of cut types to try at child nodes in the B&B tree when solving a MIP. The bit settings to enable cuts at child nodes are the same as those used to enable cuts at the root node. The default is 20478.
LS_DPARAM_MIP_CUTTIMLIM	Environment, Model	This controls the total time to be spent in cut generation throughout the solution of a MIP. Range for possible values is [0,inf). The default value is 0, indicating that no time limits will be imposed when generating cuts.
LS_IPARAM_MIP_CUTFREQ	Environment, Model	This controls the frequency of invoking cut generation at child nodes. Range for possible values is [0,inf). The default value is 10, indicating that the MIP solver will try to generate cuts at every 10 nodes.
LS_IPARAM_MIP_CUTDEPTH	Environment, Model	This controls a threshold value for the depth of nodes in the B&B tree, so cut generation will be less likely at those nodes deeper than this threshold. Range for possible values is [0,inf). The default is 5.
LS_DPARAM_MIP_LBIGM	Environment, Model	This refers to the Big-M value used in linearizing nonlinear expressions. Range for possible values is (0,inf). The default value is 1.0e+5.
LS_DPARAM_MIP_DELTA	Environment, Model	This refers to a near-zero value used in linearizing nonlinear expressions. Range for possible values is (0,inf). The default value is 1.0e-6.

LS_IPARAM_MIP_BRANCH_PRIO	Environment, Model	This controls how variable selection priorities are set and used. Possible values are: 0: If user has specified priorities, then use them. Otherwise, let LINDO API decide. 1: If user has specified priorities, then use them. However, also allow overwriting user's choices if necessary. 2: If user has specified priorities, then use them. Otherwise, do not use any priorities. 3: Let LINDO API set the priorities and ignore any user specified priorities. 4: Binaries always have higher priority over general integers. The default is 0.
LS_IPARAM_MIP_SCALING_BOUND	Environment, Model	This controls the maximum difference between the upper and lower bounds of an integer variable that will enable the scaling in the simplex solver when solving a sub problem in the branch-and-bound tree. Range for possible values is [-1,inf). The default value is 10000.
LS_IPARAM_MIP_MAXCUTPASS_TOP	Environment, Model	This controls the number passes to generate cuts on the root node. Each of these passes will be followed by a re-optimization and a new batch of cuts will be generated at the new solution. Range for possible values is [0,inf). The default value is 100.
LS_IPARAM_MIP_MAXCUTPASS_TREE	Environment, Model	This controls the number passes to generate cuts on the child nodes. Each of these passes will be followed by a re-optimization and a new batch of cuts will be generated at the new solution. Range for possible values is [0,inf). The default value is 2.
LS_IPARAM_MIP_MAXNONIMP_CUTPASS	Environment, Model	This controls the maximum number of passes allowed in cut-generation that does not improve the current relaxation. Range for possible values is [0,inf). The default value is 3.

LS_DPARAM_MIP_ADDCUTOBJTOL	Environment, Model	This specifies the minimum required change in the objective function for the cut generation phase to continue generating cuts. Range for possible values is [0,1]. The default, based on empirical testing, is set at 1.5625e-5.
LS_DPARAM_MIP_HEUMINTIMLIM	Environment, Model	This specifies the minimum time in seconds to be spent in finding heuristic solutions to the MIP model. LS_IPARAM_MIP_HEULEVEL (below) controls the heuristic used to find the integer solution. Range for possible values is [0,inf). The default is 0.
LS_DPARAM_MIP_REDCOSTFIX_CUTOFF	Environment, Model	This specifies the cutoff value as a percentage of the reduced costs to be used in fixing variables when using the *reduced cost fixing* heuristic. Range for possible values is [0,9.9e-1]. The default is 0.99.
LS_DPARAM_MIP_ADDCUTPER	Environment, Model	This determines how many constraint cuts can be added as a percentage of the number of original rows in an integer programming model. Range for possible values is [0,100). 0.75 is the default value, which means the total number of constraint cuts LINDO API adds will not exceed 50% of the original row count.
LS_DPARAM_MIP_ADDCUTPER_TREE	Environment, Model	This determines how many constraint cuts can be added at child nodes as a percentage of the number of original rows in an integer programming model. Range for possible values is [0,100). 0.75 is the default value, which means the total number of constraint cuts LINDO API adds will not exceed 50% of the original row count.
LS_DPARAM_MIP_AOPTTIMLIM	Environment, Model	This is the time in seconds beyond which the relative optimality tolerance, LS_DPARAM_MIP_PEROPTTOL, will be applied. Range for possible values is [-1,inf). The default value is 100 seconds.

LS_IPARAM_MIP_BRANCHDIR	Environment, Model	This specifies the direction to branch first when branching on a variable. Possible values are: 0: Solver decides (default), 1: Always branch up first, 2: Always branch down first.
LS_DPARAM_MIP_INTTOL	Environment, Model	An integer variable is considered integer feasible if the absolute difference from the nearest integer is smaller than this. Range for possible values is (0,0.5). The default value is 0.000001. Note, this is similar to the tolerance LS_DPARAM_MIP_RELINTTOL, but it uses absolute differences rather than relative differences.
LS_IPARAM_MIP_KEEPINMEM	Environment, Model	If this is set to 1, the integer pre-solver will try to keep LP bases in memory. This typically gives faster solution times, but uses more memory. Setting this parameter to 0 causes the pre-solver to erase bases from memory. The default is 1.
LS_DPARAM_MIP_ABSOPTTOL	Environment, Model	This is the MIP absolute optimality tolerance. Solutions must beat the incumbent by at least this absolute amount to become the new, best solution. Range for possible values is [0,inf). The default value is 0.
LS_DPARAM_MIP_RELOPTTOL	Environment, Model	This is the MIP relative optimality tolerance. Solutions must beat the incumbent by at least this relative amount to become the new, best solution. Range for possible values is (0,1). The default value is 0.000001.
LS_DPARAM_MIP_PEROPTTOL	Environment, Model	This is the MIP relative optimality tolerance that will be in effect after T seconds following the start. The value T should be specified using the LS_DPARAM_MIP_AOPTTIMLIM parameter. Range for possible values is (0,1). The default value is 0.00001.

LS_IPARAM_MIP_HEULEVEL	Environment, Model	This specifies the heuristic used to find the integer solution. Possible values are: 0: No heuristic is used. 1: A simple heuristic is used. Typically, this will find integer solutions only on problems with a certain structure. However, it tends to be fast. >2: This is an advanced heuristic that tries to find a "good" integer solution fast. In general, a value of 2 will not increase the total solution time and will find an integer solution fast on many problems. A higher value may find an integer solution faster, or an integer solution where none would have been found with a lower level. Try level 3 or 4 on "difficult" problems where 2 does not help. Higher values cause more time to be spent in the heuristic. The value may be set arbitrarily high. However, >20 is probably not worthwhile. The default is 3. LS_DPARAM_MIP_HEUMINTIMLIM (above) controls the time to be spent in searching heuristic solutions.
LS_IPARAM_MIP_SOLVERTYPE	Environment, Model	This specifies the optimization method to use when solving mixed-integer models. Possible values are: 0: Solver decides (default). 1: Use B&B only. 2: Use Enumeration and Knapsack solver only.

LS_IPARAM_MIP_NODESELRULE	Environment, Model	This specifies the node selection rule for choosing between all active nodes in the branch-and-bound tree when solving integer programs. Possible selections are:
		0: Solver decides (default).
		1: Depth first search.
		2: Choose node with worst bound.
		3: Choose node with best bound.
		4: Start with best bound. If no improvement in the gap between best bound and best integer solution is obtained for some time, switch to:
		if (number of active nodes<10000)
		Best estimate node selection (5).
		else
		Worst bound node selection (2).
		5: Choose the node with the best estimate, where the new objective estimate is obtained using pseudo costs.
		6: Same as (4), but start with the best estimate.

LS_IPARAM_MIP_BRANCHRULE	Environment, Model	This specifies the rule for choosing the variable to branch on at the selected node. Possible selections are: 0: Solver decides (default). 1: Basis rounding with pseudo reduced costs. 2: Maximum infeasibility. 3: Pseudo reduced costs only.

LS_IPARAM_MIP_PRELEVEL	Environment, Model	This controls the amount and type of MIP pre-solving at root node. Possible options are: Simple pre-solving +2 Probing +4 Coefficient reduction +8 Elimination +16 Dual reductions +32 Use dual information +64 Binary row presolving +128 Row aggregation +256 Coefficient lifting +512 Maximum pass +1024 The default value is: 2046= 2+4+8+16+32+64+128+256+512+1024.
LS_IPARAM_MIP_PREPRINTLEVEL	Environment, Model	This specifies the trace print level for the MIP presolver. Possible selections are: 0: Do not print anything (default). 1: Print summary of preprocessing.
LS_IPARAM_MIP_PRINTLEVEL	Environment, Model	This specifies the amount of printing to do. Possible values are: 0: Do not print anything. 1: Print most basic information for branch-and-bound iterations. 2: Level 1 plus print information regarding addition of cuts, etc (default).
LS_DPARAM_MIP_CUTOFFOBJ	Environment, Model	If this is specified, then any part of the branch-and-bound tree that has a bound worse than this value will not be considered. This can be used to reduce the running time if a good bound is known. Set to a large positive value (LS_INFINITY) to disable if a finite value had been specified. Range for possible values is (-inf,inf). Default is LS_INFINITY.

LS_IPARAM_MIP_USECUTOFFOBJ	Environment, Model	This is a flag for the parameter LS_DPARAM_MIP_CUTOFFOBJ. The value of 0 means that the current cutoff value is ignored, else it is used as defined. If you don't want to lose the value of the parameter LS_DPARAM_MIP_CUTOFFOBJ, this provides an alternative to disabling the cutoff objective. Possible values are 0 (no), 1 (yes) and -1 (the solver decides). The default is 0.
LS_DPARAM_MIP_RELINTTOL	Environment, Model	An integer variable is considered integer feasible if the difference between its value and the nearest integer value divided by the value of the nearest integer is less than this. Range for possible values is (0,0.5). The default value is 0.000008. Note this is a relative version of the LS_DPARAM_MIP_INTTOL tolerance.
LS_IPARAM_MIP_REOPT	Environment, Model	This specifies which optimization method to use when doing reoptimization from a given basis. Possible values are: 0: Solver decides (default). 1: Use primal method. 2: Use dual simplex. 3: Use barrier solver (with or without basis crossover, depending on LS_IPARAM_SOLVER_IPMSOL parameter setting).

LS_IPARAM_MIP_STRONGBRANCHLEVEL	Environment, Model	This specifies the depth from the root in which strong branching is used. Range for possible values is [0,inf). The default value of 10 means that strong branching is used on a level of 1 to 10 measured from the root. Strong branching finds the real bound for branching on a given variable, which, in most cases, requires a solution of a linear program and may therefore also be quite expensive in computing time. However, if used on nodes close to the root node of the tree, it also gives a much better bound for that part of the tree and can therefore reduce the size of the branch-and-bound tree.
LS_IPARAM_MIP_TREEREORDERLEVEL	Environment, Model	This specifies the tree reordering level. Range for possible values is [0,inf). The default is 10.
LS_IPARAM_MIP_ANODES_SWITCH_DF	Environment, Model	This specifies the threshold on active nodes for switching to depth-first search rule. Range for possible values is [-1,inf). The default is 50,000.
LS_IPARAM_MIP_SWITCHFAC_SIM_IPM	Environment, Model	This specifies the (positive) factor that multiplies the number of constraints to impose an iteration limit to simplex method and trigger a switch over to the barrier method. Range for possible values is [-1,inf). The default value is –1, which means that no iteration limit is imposed.

LS_DPARAM_MIP_TIMLIM	Environment, Model	This is the time limit in seconds for MIP solver. Range for possible values is [-1.0, inf). The default value is –1, which means no time limit is imposed. However, the value of LS_IPARAM_SOLVER_TIMLMT will be applied to each continuous sub problem solve. If the value of this parameter is greater than 0, then the value of LS_IPARAM_SOLVER_TIMLMT will be disregarded. If the time limit, LS_DPARAM_MIP_TIMLIM, is reached and a feasible integer solution was found, it will be installed as the incumbent (best known) solution.
LS_IPARAM_MIP_ITRLIM	Environment, Model	This is the total LP iteration limit summed over all branches for branch-and-bound. Range for possible values is [-1,inf). The default value is –1, which means no iteration limit is imposed. If this iteration limit is reached, branch-and-bound will stop and the best feasible integer solution found will be installed as the incumbent (best known) solution.
LS_IPARAM_MIP_BRANCH_LIMIT	Environment, Model	This is the limit on the total number of branches to be created during branch-and-bound. Range for possible values is [-1,inf). The default value is –1, which means no limit is imposed. If the branch limit, LS_IPARAM_MIP_BRANCH_LIMIT, is reached and a feasible integer solution was found, it will be installed as the incumbent (best known) solution.

LS_IPARAM_MIP_TOPOPT	Environment, Model	This specifies which optimization method to use when there is no previous basis. Possible values are: 0: Solver decides (default). 1: Use primal method. 2: Use dual simplex. 3: Use barrier solver (with or without basis crossover, depending on LS_IPARAM_SOLVER_IPMSOL parameter setting described above)
LS_DPARAM_MIP_LSOLTIMLIM	Environment, Model	This value controls the time limit until finding a new integer solution since the last integer solution found. Range for possible values is [-1,inf). The default value is -1, which means no time limit is imposed.
LS_IPARAM_MIP_DUAL_SOLUTION	Environment, Model	This flag controls whether the dual solution to the LP relaxation that yielded the optimal MIP solution will be computed or not. Possible values are 0 (no), 1 (yes). The default is 0.
LS_IPARAM_MIP_AGGCUTLIM_TOP	Environment, Model	This specifies an upper limit on the number of constraints to be involved in the derivation of an aggregation cut at the root node. Range for possible values is [-1,inf). The default is –1, which means that the solver will decide.
LS_IPARAM_MIP_AGGCUTLIM_TREE	Environment, Model	This specifies an upper limit on the number of constraints to be involved in the derivation of an aggregation cut at the tree nodes. Range for possible values is [-1,inf). The default is 3.
LS_DPARAM_MIP_MINABSOBJSTEP	Environment, Model	This specifies the value to update the cutoff value each time a mixed integer solution is found. Range for possible values is (-inf,inf). The default is 0.0

LS_IPARAM_MIP_PSEUDOCOST_RULE	Environment, Model	This specifies the rule in pseudocost computations for variable selection. Possible values are 0: solver decides (default). 1: only use min pseudo cost. 2: only use max pseudo cost. 3: use quadratic score function and the pseudo cost weigth. 4: same as 3 without quadratic score.
LS_IPARAM_MIP_ENUM_HEUMODE	Environment, Model	This specifies the frequency of enumeration heuristic. Possible values are 0: off 1: only at top (root) node without cuts. 2: both at top (root) and tree nodes without cuts. 3: same as 1 with cuts. 4: same as 2 with cuts (default).
LS_IPARAM_MIP_PRELEVEL_TREE	Environment, Model	This controls the amount and type of MIP pre-solving at tree nodes. Possible options are: Simple pre-solving +2 Probing +4 Coefficient reduction +8 Elimination +16 Dual reductions +32 Use dual information +64 Binary row presolving +128 Row aggregation +256 Maximum pass +512 The default value is: 686 = 2+4+8+32+128+512.
LS_DPARAM_MIP_PSEUDOCOST_WEIGT	Environment, Model	This specifies the weight in pseudocost computations for variable selection. Range for possible values is (0,inf). The default is 6.25.
LS_DPARAM_MIP_REDCOSTFIX_CUTOFF_TREE	Environment, Model	This specifies the cutoff value as a percentage of the reduced costs to be used in fixing variables when using the *reduced cost fixing* heuristic at tree nodes. Range for possible values is [0,9.9e-1]. The default is 0.9.
LS_DPARAM_MIP_OBJ_THRESHOLD	Environment, Model	Reserved for future use.

LS_IPARAM_MIP_LOCALBRANCHNUM	Environment, Model	
LS_DPARAM_MIP_SWITCHFAC_SIM_IPM_TIME	Environment, Model	This specifies the (positive) factor that multiplies the number of constraints to impose a time limit to simplex method and trigger a switch over to the barrier method. Range for possible values is [-1.0,inf). The default value is –1.0, which means that no time limit is imposed.
LS_DPARAM_MIP_ITRLIM	Environment, Model	This is the total LP iteration limit (stored as a double variable) summed over all branches for branch-and-bound. Range for possible values is [-1,inf). The default value is –1, which means no iteration limit is imposed. If this iteration limit is reached, branch-and-bound will stop and the best feasible integer solution found will be installed as the incumbent (best known) solution.
LS_IPARAM_MIP_MAXNUM_MIP_SOL_STORAGE	Environment, Model	This specifies the maximum number of k-best solutions to store. Possible values are positive integers. Default is 1.

Global Optimization Parameters

LS_DPARAM_GOP_OPTTOL	Environment, Model	This value is the GOP optimality tolerance. Solutions must beat the incumbent by at least this amount to become the new best solution. Range for possible values is [0,1]. The default value is 1.0e-6.
LS_DPARAM_GOP_FLTTOL	Environment, Model	This value is the GOP floating-point tolerance. It specifies the maximum rounding errors in the floating-point computation. Range for possible values is [0,1]. The default value is 1.0e-10.
LS_DPARAM_GOP_BOXTOL	Environment, Model	This value specifies the minimal width of variable intervals in a box allowed to branch. Range for possible values is [0,1]. The default value is 1.0e-6.

LS_DPARAM_GOP_WIDTOL	Environment, Model	This value specifies the maximal width of variable intervals for a box to be considered as an incumbent box containing an incumbent solution. It is used when LS_IPARAM_GOP_MAXWIDMD is set at 1. Range for possible values is [0,1]. The default value is 1e-4.
LS_DPARAM_GOP_DELTATOL	Environment, Model	This value is the delta tolerance in the GOP convexification. It is a measure of how closely the additional constraints added as part of convexification should be satisfied. Range for possible values is [0,1]. The default value is 1e-7.
LS_DPARAM_GOP_BNDLIM	Environment, Model	This value specifies the maximum magnitude of variable bounds used in the GOP convexification. Any lower bound smaller than the negative of this value will be treated as the negative of this value. Any upper bound greater than this value will be treated as this value. This helps the global solver focus on more productive domains. Range for possible values is [0,inf). The default value is 1e10.
LS_IPARAM_GOP_TIMLIM	Environment, Model	This is the integer time limit in seconds for GOP branch-and-bound. Range for possible values is [-1, INT_MAX). The default value is -1, which means no time limit is imposed.

LS_IPARAM_GOP_OPTCHKMD	Environment, Model	This specifies the criterion used to certify the global optimality. Possible values are 0: the absolute deviation of objective lower and upper bounds should be smaller than LS_DPARAM_GOP_OPTTOL at the global optimum. 1: the relative deviation of objective lower and upper bounds should be smaller than LS_DPARAM_GOP_OPTTOL at the global optimum. 2: which means both absolute and relative tolerances are satisfied at global optimum (default).
LS_IPARAM_GOP_MAXWIDMD	Environment, Model	This is the maximum width flag for the global solution. The GOP branch-and-bound may continue contracting a box with an incumbent solution until its maximum width is smaller than LS_DPARAM_GOP_WIDTOL. The possible value are 0: the maximum width criterion is suppressed (default). 1: the maximum width criterion is performed.

LS_IPARAM_GOP_BRANCHMD	Environment, Model	This specifies the direction to branch first when branching on a variable. The branch variable is selected as the one that holds the largest magnitude in the measure. Possible values are: 0: Absolute width. 1: Locally relative width. 2: Globally relative width. 3: Globally relative distance from the convex minimum to the bounds. 4: Absolute violation between the function and its convex envelope at the convex minimum. 5: Relative violation between the function and its convex envelope at the convex minimum. The default value is 5.
LS_IPARAM_GOP_PRELEVEL	Environment, Model	This controls the amount and type of GOP pre-solving. Possible options are: • Initial local optimization +2 • Initial linear constraint propagation +4 • Recursive linear constraint propagation +8 • Recursive nonlinear constraint propagation +16 • Search for good near feasible solutions. +32 • Check for unboundedness +64 • Alter derivative methods +128 • MIP pre-optimizations +256 • NLP pre-optimizations +512 The default value is 1022 = 2+4+8+16+32+64+128+256+512
LS_IPARAM_GOP_POSTLEVEL	Environment, Model	This controls the amount and type of GOP post-solving. Possible options are: Apply *LSgetBestBound*() +2 Reoptimize variable bounds +4 The default value is: 6 = 2+4

LS_IPARAM_GOP_BBSRCHMD	Environment, Model	This specifies the node selection rule for choosing between all active nodes in the GOP branch-and-bound tree when solving global optimization programs. Possible selections are: 0: Depth first search. 1: Choose node with worst bound. The default value is 1.
LS_IPARAM_GOP_DECOMPPTMD	Environment, Model	This specifies the decomposition point selection rule. In the branch step of GOP branch-and-bound, a branch point *M* is selected to decompose the selected variable interval [*Lb*, *Ub*] into two sub-intervals, [*Lb*, *M*] and [*M*, *Ub*]. Possible options are: 0: mid-point. 1: local minimum/convex minimum. The default value is 1.
LS_IPARAM_GOP_ALGREFORMMD	Environment, Model	This controls the algebraic reformulation rule for a GOP. The algebraic reformulation and analysis is very crucial in building a tight convex envelope to enclose the nonlinear/non-convex functions. A lower degree of overestimation on convex envelopes helps increase the convergence rate to the global optimum. Possible options are: Rearrange and collect terms +2 Expand all parentheses +4 Retain nonlinear functions +8 Selectively expand parentheses +16 The default value is: 18 = 2+16
LS_IPARAM_GOP_PRINTLEVEL	Environment, Model	This specifies the amount of print to do for the global solver. Possible selections are: 0: Do not print anything. 1: Print information for GOP branch-and-bound iterations (default).

LS_IPARAM_GOP_RELBRNDMD	Environment, Model	This controls the reliable rounding rule in the GOP branch-and-bound. The global solver applies many sub-optimizations to estimate the lower and upper bounds on the global optimum. A rounding error or numerical instability could unintentionally cut off a good solution. A variety of reliable approaches are available to improve the precision. • Use smaller optimality/feasibility tolerances and appropriate pre-solving options +2 • Apply interval arithmetic to re-verify the solution feasibility +4 The default value is 0.
LS_IPARAM_GOP_BNDLIM_MODE	Environment, Model	This value is associated with the parameter LS_DPARAM_GOP_BNDLIM and determines the mode how the specified bound limit will be used. Possible values are: 0:Do not use the bound limit on the variables. 1: Use the bound limit right at the beginning of global optimization. 2: Use the bound limit after the initial local optimization, if selected. This properly sets the bound limit for each variable to include the initial solution, if any, within the range.
LS_IPARAM_GOP_OPT_MODE	Environment, Model	This specifies the mode for global search. Possible values are 0: global search for a feasible solution (thus a feasibility certificate). 1: global search for an optimal solution (default). 2: global search for an unboundedness certificate. The default value is 1.

LS_IPARAM_GOP_BRANCH_LIMIT	Environment, Model	This is the integer limit on the total number of branches to be created during branch-and-bound in GOP tree. Range for possible values is [-1, INT_MAX).The default value is –1, which means no limit is imposed. If the branch limit, LS_IPARAM_GOP_BRANCH_LIMIT, is reached and a feasible solution was found, it will be installed as the incumbent (best known) solution.

LS_IPARAM_GOP_CORELEVEL	Environment, Model	This controls the strategy of GOP branch-and-bound procedure. Possible options are: - LP convex relaxation +2 - Honor NLP solutions +4 - Box Branching +8 - Honor IPM solutions +16 The default is 30.
LS_IPARAM_GOP_HEU_MODE	Environment, Model	This specifies the heuristic used in the global solver to find good solution. Possible values are: 0: No heuristic is used. 1: A simple heuristic is used. Typically, this will put more efforts in searching for good solutions, and less in bound tightening. The default is 0.
LS_IPARAM_GOP_SUBOUT_MODE	Environment, Model	This is a flag indicating whether fixed variables are substituted out of the instruction list used in the global solver. Possible values are 0 (no), 1 (yes). The default is 1.
LS_IPARAM_GOP_USE_NLPSOLVE	Environment, Model	This is reserved for internal use only.
LS_IPARAM_GOP_LSOLBRANLIM	Environment, Model	This value controls the branch limit until finding a new nonlinear solution since the last nonlinear solution is found. Range for possible values is [-1,inf). The default value is –1, which means no branch limit is imposed.
LS_IPARAM_GOP_LPSOPT	Environment, Model	This is reserved for internal use only.
LS_DPARAM_GOP_TIMLIM	Environment, Model	This is the time limit in seconds for GOP branch-and-bound. Range for possible values is [-1.0,inf). The default value is -1.0, which means no time limit is imposed.

LS_DPARAM_GOP_BRANCH_LIMIT	Environment, Model	This is the limit on the total number of branches (stored as a double) to be created during branch-and-bound in GOP tree. Range for possible values is [-1, +inf).The default value is –1, which means no limit is imposed. If the branch limit, LS_DPARAM_GOP_BRANCH_LIMIT , is reached and a feasible solution was found, it will be installed as the incumbent (best known) solution.
LS_IPARAM_GOP_QUADMD	Environment, Model	Reserved for future use.

License Information Parameters

LS_IPARAM_LIC_PLATFORM	Environment	This returns the platform identifier for a given license key. This value cannot be set.
LS_IPARAM_LIC_CONSTRAINTS	Environment	This returns an integer containing the number of constraints allowed for a single model. It returns -1 if the number is unlimited. This value cannot be set.
LS_IPARAM_LIC_VARIABLES	Environment	This returns an integer containing the maximum number of variables allowed in a single model. It returns -1 if the number is unlimited. This value cannot be set.
LS_IPARAM_LIC_INTEGERS	Environment	This returns an integer containing the maximum number of integer variables allowed in a single model. It returns -1 if the number is unlimited. This value cannot be set.
LS_IPARAM_LIC_NONLINEARVARS	Environment	This returns an integer containing the maximum number of nonlinear variables allowed in a single model. It returns -1 if the number is unlimited. This value cannot be set.
LS_IPARAM_LIC_GOP_INTEGERS	Environment	This returns an integer containing the maximum number of integer variables allowed in a global optimization model. It returns -1 if the number is unlimited. This value cannot be set.
LS_IPARAM_LIC_GOP_NONLINEARVARS	Environment	This returns an integer containing the maximum number of nonlinear variables allowed in a global optimization model. It returns -1 if the number is unlimited. This value cannot be set.
LS_IPARAM_LIC_DAYSTOEXP	Environment	This returns an integer containing the number of days until the license expires. It returns -2 if there is no expiration date. This value cannot be set.
LS_IPARAM_LIC_DAYSTOTRIALEXP	Environment	This returns an integer containing the number of days until the trial features of the license expires. It returns -2 if there is no trial period. This value cannot be set.

LS_IPARAM_LIC_BARRIER	Environment	This returns an integer containing a 1 if the barrier solver option is available and 0 if it is not. The barrier solver, also known as the “interior point” solver, tends to be faster on some large models. A license for the barrier solver may be obtained through LINDO Systems. This value cannot be set.
LS_IPARAM_LIC_NONLINEAR	Environment	This returns an integer containing a 1 if the nonlinear solver option is available and 0 if it is not. A license for the nonlinear solver may be obtained through LINDO Systems. This value cannot be set.
LS_IPARAM_LIC_GLOBAL	Environment	This returns an integer containing a 1 if the global solver option is available and 0 if it is not. A license for the global solver may be obtained through LINDO Systems. This value cannot be set.
LS_IPARAM_LIC_EDUCATIONAL	Environment	This returns an integer containing a 1 or a 0. 1 means that the current license is for educational use only. This value cannot be set.
LS_IPARAM_LIC_NUMUSERS	Environment	This returns an integer specifying the maximum number of concurrent users allowed to use the current license. This value cannot be set.
LS_IPARAM_LIC_RUNTIME	Environment	This returns an integer containing a 1 or a 0. 1 meaning the license is for runtime use only. This value cannot be set.

Model Analysis Parameters

LS_IPARAM_IIS_METHOD	Model	This specifies the method to use in analyzing infeasible models to locate an IIS. Possible values are: LS_IIS_DEFAULT = 0, LS_IIS_DEL_FILTER =1, LS_IIS_ADD_FILTER =2, LS_IIS_GBS_FILTER =3, LS_IIS_DFBS_FILTER =4, LS_IIS_FSC_FILTER =5, LS_IIS_ELS_FILTER =6 The default is LS_IIS_DEFAULT
LS_IPARAM_IIS_USE_EFILTER	Model	This flag controls whether the Elastic Filter should be enabled as the supplementary filter in analyzing infeasible models when the Elastic Filter is not the primary method. Possible values are 0 (no), 1 (yes) and -1 (the solver decides). The default is 0.
LS_IPARAM_IIS_USE_GOP	Model	This flag controls whether the global optimizer should be enabled in analyzing infeasible NLP models. Possible values are 0 (no), 1 (yes) and -1 (the solver decides). The default is 0.
LS_IPARAM_IIS_ANALYZE_LEVEL	Model	This controls the level of analysis when locating an IIS to debug an infeasible model. Bit mask values are: LS_NECESSARY_ROWS= 1, LS_NECESSARY_COLS = 2, LS_SUFFICIENT_ROWS= 4, LS_SUFFICIENT_COLS = 8. The default is 1;
LS_IPARAM_IUS_ANALYZE_LEVEL	Model	This controls the level of analysis when locating an IUS to debug an unbounded LP. Bit mask values are: LS_NECESSARY_COLS= 2, LS_SUFFICIENT_COLS = 8. The default is 2.

LS_IPARAM_IUS_TOPOPT	Environment, Model	This specifies which optimization method to use when there is no previous basis during the analysis. Possible values are: 0: Solver decides (default). 1: Use primal method. 2: Use dual simplex. 3: Use barrier solver (with or without basis crossover, depending on LS_IPARAM_SOLVER_IPMSOL parameter setting described above)
LS_IPARAM_IIS_REOPT	Environment, Model	This specifies which optimization method to use when starting from a given basis. Possible values are: 0: Solver decides (default). 1: Use primal method. 2: Use dual simplex. 3:Use barrier solver (with or without basis crossover, depending on LS_IPARAM_SOLVER_IPMSOL parameter setting).
LS_IPARAM_IIS_USE_SFILTER	Environment, Model	This is a flag indicating is sensitivity filter will be used during IIS search. Possible values are 0 (no), 1 (yes) and -1 (the solver decides). The default is 1.
LS_IPARAM_IIS_PRINT_LEVEL	Environment, Model	This specifies the amount of print to do during IIS search. Possible values are: 0: Do not print anything (default). >0: Print more information. Default is 2.
LS_IPARAM_IIS_INFEAS_NORM	Environment, Model	This specifies the norm to measure infeasibilities in IIS search. Possible values are: • IIS_NORM_NONE (default) : Fast scan without measuring infeasibilities. • IIS_NORM_ONE: Use L-1 norm. • IIS_NORM_INFINITY: Use L-infinity norm.
LS_IPARAM_IIS_ITER_LIMIT	Environment, Model	This is the iteration limit for IIS search. The default value is -1, which means no iteration limit is imposed.

LS_IPARAM_IIS_TIME_LIMIT	Environment, Model	This is the time limit for IIS search. The default value is -1, which means no time limit is imposed.

Stochastic Parameters

LS_IPARAM_STOC_SAMPSIZE_SPAR	Model	Common sample size per stochastic parameter. Possible values are positive integers. (reserved for future use)
LS_IPARAM_STOC_SAMPSIZE_NODE	Model	Common sample size per stage (reserved for future use).
LS_IPARAM_STOC_RG_SEED	Model	Seed to initialize the random number generator. Possible values are positive integers. The default is 1031.
LS_IPARAM_STOC_METHOD	Model	Stochastic optimization method to solve the model. Possible values are: • LS_METHOD_STOC_FREE • LS_METHOD_STOC_DETEQ • LS_METHOD_STOC_NBD • LS_METHOD_STOC_ALD The default is LS_METHOD_STOC_FREE
LS_IPARAM_STOC_REOPT	Model	Reoptimization method to solve the node-models. Possible values are: • LS_METHOD_FREE (default) • LS_METHOD_PSIMPLEX • LS_METHOD_DSIMPLEX • LS_METHOD_BARRIER • LS_METHOD_NLP
LS_IPARAM_STOC_TOPOPT	Model	Optimization method to solve the root problem. Possible values are: • LS_METHOD_FREE (default) • LS_METHOD_PSIMPLEX • LS_METHOD_DSIMPLEX • LS_METHOD_BARRIER • LS_METHOD_NLP
LS_IPARAM_STOC_ITER_LIM	Model	Iteration limit for stochastic solver. Possible values are positive integers or (-1) no limit. Default is -1.
LS_IPARAM_STOC_PRINT_LEVEL	Model	Print level to display progress information during optimization. Possible values are nonnegative integers. Default is 0.

LS_IPARAM_STOC_DETEQ_TYPE	Model	Type of deterministic equivalent to be used by the solver. Possible values are: • LS_DETEQ_FREE (default) • LS_DETEQ_IMPLICIT • LS_DETEQ_EXPLICIT LS_DETEQ_IMPLICT is valid for linear and integer models only.
LS_IPARAM_STOC_CALC_EVPI	Model	Flag to enable/disable calculation of lower bounds on EVPI. Possible values are (0): disable, (1) enable. Default is 1.
LS_IPARAM_STOC_DEBUG_LEVEL	Model	Debug level (for internal use).
LS_IPARAM_STOC_SAMP_CONT_ONLY	Model	Flag to restrict sampling to continuous stochastic parameters only or not. Possible values are (0): disable, (1) enable. Default is 1.
LS_IPARAM_STOC_BUCKET_SIZE	Model	Bucket size in Benders decomposition. Possible values are positive integers or (-1) for solver decides. Default is -1.
LS_IPARAM_STOC_MAX_NUMSCENS	Model	Maximum number of scenarios before forcing automatic sampling. Possible values are positive integers. Default is 40,000.
LS_IPARAM_STOC_SHARE_BEGSTAGE	Model	Stage beyond which node-models share the same model structure. Possible values are positive integers less than or equal to number of stages in the model or (-1) for solver decides. Default is -1.
LS_IPARAM_STOC_NODELP_PRELEVEL	Model	Presolve level solving node-models. Possible values are bitmasks defined in LS_IPARAM_LP_PRELEVEL. Default is the default LP presolve level.
LS_DPARAM_STOC_TIME_LIM	Model	Time limit for stochastic solver. Possible values are nonnegative real numbers or -1.0 for solver decides. Default is -1.0.
LS_DPARAM_STOC_RELOPTTOL	Model	Relative optimality tolerance (w.r.t lower and upper bounds on the true objective) to stop the solver. Possible values are reals in (0,1) interval. Default is 1e-7.

LS_DPARAM_STOC_ABSOPTTOL	Model	Absolute optimality tolerance (w.r.t lower and upper bounds on the true objective) to stop the solver. . Possible values are reals in (0,1) interval. Default is 1e-7.
LS_IPARAM_STOC_VARCONTROL_METHOD	Model	Sampling method for variance reduction. Possible values are: • LS_MONTECARLO • LS_LATINSQUARE • LS_ANTITHETIC • LS_LATINSQUARE + LS_ANTITHETIC LS_MONTECARLO implies the use of standard sampling with no variance reduction. LS_ANTITHETIC implies the use of antithetic pairs of uniform variates to control variance. LS_LATINSQUARE implies the use of basic Latin-hypercude sampling which is known to be efficient for most distributions. Default is LS_LATINSQUARE.
LS_IPARAM_STOC_CORRELATION_TYPE	Model	Correlation type associated with the correlation matrix. Possible values are: • LS_CORR_TARGET • LS_CORR_PEARSON • LS_CORR_KENDALL • LS_CORR_SPEARMAN Default is LS_CORR_PEARSON.
LS_IPARAM_STOC_WSBAS	Model	Warm start basis for wait-see model .
LS_IPARAM_STOC_ALD_OUTER_ITER_LIM	Model	Outer loop iteration limit for ALD.
LS_IPARAM_STOC_ALD_INNER_ITER_LIM	Model	Inner loop iteration limit for ALD.
LS_DPARAM_STOC_ALD_DUAL_FEASTOL	Model	Dual feasibility tolerance for ALD.
LS_DPARAM_STOC_ALD_PRIMAL_FEASTOL	Model	Primal feasibility tolerance for ALD.
LS_DPARAM_STOC_ALD_DUAL_STEPLEN	Model	Dual step length for ALD.
LS_DPARAM_STOC_ALD_PRIMAL_STEPLEN	Model	Primal step length for ALD.
LS_IPARAM_CORE_ORDER_BY_STAGE	Model	Order nontemporal models or not.
LS_SPARAM_STOC_FMT_NODE_NAME	Model	Node name format.
LS_SPARAM_STOC_FMT_SCENARIO_NAME	Model	Scenario name format.

LS_IPARAM_STOC_MAP_MPI2LP	Model	Flag to specify whether stochastic parameters in MPI will be mapped as LP matrix elements.

Available Information

These macros refer to available information about the model, solution or sample associated with the specified object.

General Model and Solution Information

LS_IINFO_METHOD	Model	Optimization method used.
LS_IINFO_NUM_CONES	Model	Number of cones.
LS_IINFO_NUM_CONE_NONZ	Model	Number of nonzeros in the conic structure.
LS_IINFO_LEN_CONENAMES	Model	Length of cone names.
LS_DINFO_INST_VAL_MIN_COEF	Model	Minimum coefficient in instruction list.
LS_IINFO_INST_VARNDX_MIN_COEF	Model	Variable index of the minimum coefficient.
LS_IINFO_INST_CONNDX_MIN_COEF	Model	Constraint index of the minimum coefficient.
LS_DINFO_INST_VAL_MAX_COEF	Model	Maximum coefficient in instruction list.
LS_IINFO_INST_VARNDX_MAX_COEF	Model	Variable index of the maximum coefficient.
LS_IINFO_INST_VARNDX_MAX_COEF	Model	Variable index of the maximum coefficient.
LS_IINFO_INST_CONNDX_MAX_COEF	Model	Constraint index of the maximum coefficient.
LS_IINFO_NUM_CALL_FUN	Model	Number of function evaluations.
LS_IINFO_NUM_CALL_DEV	Model	Number of first-derivative (Jacobian) evaluations.
LS_IINFO_NUM_CALL_HES	Model	Number of second-derivative (Hessian) evaluations.
LS_IINFO_ELAPSED_TIME	Model	Total CPU time elapsed solving the continuous problem.
LS_IINFO_MODEL_STATUS	Model	The status of given model based on the result of last optimization.
LS_IINFO_PRIMAL_STATUS	Model	The status of the primal model based on the result of the last optimization.
LS_IINFO_IPM_STATUS	Model	The status of the interior-point solution based on the barrier solver.
LS_IINFO_DUAL_STATUS	Model	Dual solution status.
LS_IINFO_BASIC_STATUS	Model	Basic solution status.
LS_IINFO_SIM_ITER	Model	Number of simplex iterations performed when solving a continuous problem.

LS_IINFO_BAR_ITER	Model	Number of barrier iterations performed when solving a continuous problem.
LS_IINFO_NLP_ITER	Model	Number of nonlinear iterations performed when solving a continuous problem.
LS_DINFO_POBJ	Model	Primal objective value of a continuous problem.
LS_DINFO_DOBJ	Model	Dual objective value of a continuous problem.
LS_DINFO_PINFEAS	Model	Maximum primal infeasibility.
LS_DINFO_DINFEAS	Model	Maximum dual infeasibility.
LS_DINFO_MSW_POBJ	Model	Value of the incumbent objective value when using the multistart solver.
LS_IINFO_MSW_PASS	Model	Number of multistart passes.
LS_IINFO_MSW_NSOL	Model	Number of distinct solutions found when using the multistart solver.
LS_DINFO_IPM_POBJ	Model	Primal objective value w.r.t the interior-point solution.
LS_DINFO_IPM_DOBJ	Model	Dual objective value w.r.t the interior-point solution.
LS_DINFO_IPM_PINFEAS	Model	Primal infeasibility w.r.t the interior-point solution.
LS_DINFO_IPM_DINFEAS	Model	Dual infeasibility w.r.t the interior-point solution.
LS_IINFO_NUM_CONS	Model	Number of constraints in the model.
LS_IINFO_NUM_VARS	Model	Number of variables in the model.
LS_IINFO_NUM_NONZ	Model	Number of nonzeros in the linear portion of the model.
LS_IINFO_NUM_NLP_CONS	Model	Number of NLP constraints in the model.
LS_IINFO_NUM_NLP_VARS	Model	Number of NLP variables in the model.
LS_IINFO_NUM_QC_NONZ	Model	Number of nonzeros in the quadratic matrices.
LS_IINFO_NUM_NLP_NONZ	Model	Number of nonzeros in the nonlinear portion of the model.
LS_IINFO_NUM_NLPOBJ_NONZ	Model	Number of nonzeros in the nonlinear objectives in the model.

LS_IINFO_NUM_RDCONS	Model	Number of constraints in the presolved (reduced) model.
LS_IINFO_NUM_RDVARS	Model	Number of variables in the presolved (reduced) model.
LS_IINFO_NUM_RDNONZ	Model	Number of nonzeros in the linear portion of the presolved (reduced) model.
LS_IINFO_NUM_RDINT	Model	Number of integer (including binary) variables in the presolved (reduced) model.
LS_IINFO_LEN_VARNAMES	Model	Cumulative size of the variable names in the model.
LS_IINFO_LEN_CONNAMES	Model	Cumulative size of the constraint names in the model.
LS_IINFO_NUM_BIN	Model	Number of binary variables in the model.
LS_IINFO_NUM_INT	Model	Number of general integer variables in the model.
LS_IINFO_NUM_CONT	Model	Number of continuous variables in the model.
LS_IINFO_PRE_NUM_RED	Model	Number of reductions in pre-solve.
LS_IINFO_PRE_TYPE_RED	Model	Type of last reduction.
LS_IINFO_PRE_NUM_RDCONS	Model	Number of constraints in the pre-solved model.
LS_IINFO_PRE_NUM_RDVARS	Model	Number of variables in the pre-solved model.
LS_IINFO_PRE_NUM_RDNONZ	Model	Number of nonzeros in the pre-solved model.
LS_IINFO_PRE_NUM_RDINT	Model	Number of integer variables in the pre-solved model.
LS_IINFO_NUM_SUF_ROWS	Model	Number of sufficient rows in IIS.
LS_IINFO_NUM_IIS_ROWS	Model	Number of necessary rows in IIS.
LS_IINFO_NUM_SUF_BNDS	Model	Number of sufficient variable bounds in IIS.
LS_IINFO_NUM_IIS_BNDS	Model	Number of necessary variable bounds in IIS.
LS_IINFO_NUM_SUF_COLS:	Model	Number of sufficient columns in IUS.

LS_IINFO_NUM_IUS_COLS:	Model	Number of necessary columns in IUS.
LS_IINFO_ERR_OPTIM	Model	The error code produced at last optimization session.
LS_DINFO_INST_VAL_MIN_COEF	Model	Values of the minimum matrix coefficient in the model.
LS_IINFO_INST_VARNDX_MIN_COEF	Model	Variable index of the minimum matrix coefficient in the model.
LS_IINFO_INST_CONNDX_MIN_COEF	Model	Constraint index of the minimum matrix coefficient in the model.
LS_DINFO_INST_VAL_MAX_COEF	Model	Values of the maximum matrix coefficient in the model.
LS_IINFO_INST_VARNDX_MAX_COEF	Model	Variable index of the maximum matrix coefficient in the model.
LS_IINFO_INST_CONNDX_MAX_COEF	Model	Constraint index of the maximum matrix coefficient in the model.
LS_IINFO_NUM_VARS_CARD	Model	Number of cardinality sets.
LS_IINFO_NUM_VARS_SOS1	Model	Number of type-1 SOS variables.
LS_IINFO_NUM_VARS_SOS2	Model	Number of type-2 SOS variables.
LS_IINFO_NUM_VARS_SOS3	Model	Number of type-3 SOS variables.
LS_IINFO_NUM_VARS_SCONT	Model	Number of semi-continous variables.
LS_IINFO_NUM_CONS_L	Model	Number of ‘less-than-or-equal-to’ constraints.
LS_IINFO_NUM_CONS_E	Model	Number of ‘equality’ type constraints.
LS_IINFO_NUM_CONS_G	Model	Number of ‘greater-than-or-equal-to’ type constraints.
LS_IINFO_NUM_CONS_R	Model	Number of ranged constraints.
LS_IINFO_NUM_CONS_N	Model	Number of neutral (objective) constraints.
LS_IINFO_NUM_VARS_LB	Model	Number of variables with only a lower bound.
LS_IINFO_NUM_VARS_UB	Model	Number of variables with only an upper bound.
LS_IINFO_NUM_VARS_LUB	Model	Number of variables with both lower and upper bounds.
LS_IINFO_NUM_VARS_FR	Model	Number of free variables.
LS_IINFO_NUM_VARS_FX	Model	Number of fixed variables.

Integer Optimization Information

LS_DINFO_MIP_OBJ	Model	MIP objective value.
LS_DINFO_MIP_BESTBOUND	Model	Best bound on MIP objective.
LS_DINFO_MIP_TOT_TIME	Model	Total CPU time spent for solving a MIP.
LS_DINFO_MIP_OPT_TIME	Model	CPU time spent for optimizing the MIP.
LS_DINFO_MIP_HEU_TIME	Model	CPU time spent in MIP presolver and other heuristics.
LS_IINFO_MIP_LPCOUNT	Model	Number of LPs solved for solving a MIP.
LS_IINFO_MIP_BRANCHCOUNT	Model	Number of branches generated for solving a MIP.
LS_IINFO_MIP_ACTIVENODES	Model	Number of remaining nodes to be explored.
LS_IINFO_MIP_LTYPE	Model	Step at which the last integer solution was found during the optimization of a MIP. Possible values are: 10: backward strong branching or tree reordering 9: simple enumerator 8: advanced branching 7: advanced heuristics 6: after adding cuts 5: on the top 4: simple rounding heuristic 3: strong branching 2: knapsack solver or enumerator 1: normal branching
LS_IINFO_MIP_AOPTTIMETOSTOP	Model	Time to approximate optimality.
LS_IINFO_MIP_STATUS	Model	Status of MIP solution.
LS_IINFO_MIP_SIM_ITER	Model	Number of simplex iterations performed when solving a MIP.
LS_IINFO_MIP_BAR_ITER	Model	Number of barrier iterations performed when solving a MIP.
LS_IINFO_MIP_NLP_ITER	Model	Number of nonlinear iterations performed for solving a MIP.

LS_IINFO_MIP_NUM_TOTAL_CUTS	Model	Number of total cuts generated.
LS_IINFO_MIP_GUB_COVER_CUTS	Model	Number of GUB cover cuts generated.
LS_IINFO_MIP_FLOW_COVER_CUTS	Model	Number of flow cover cuts generated.
LS_IINFO_MIP_LIFT_CUTS	Model	Number of lifted knapsack covers generated.
LS_IINFO_MIP_PLAN_LOC_CUTS	Model	Number of plant location cuts generated.
LS_IINFO_MIP_DISAGG_CUTS	Model	Number of disaggregation cuts generated.
LS_IINFO_MIP_KNAPSUR_COVER_CUTS	Model	Number of surrogate knapsack covers generated.
LS_IINFO_MIP_LATTICE_CUTS	Model	Number of lattice cuts generated.
LS_IINFO_MIP_GOMORY_CUTS	Model	Number of Gomory cuts generated.
LS_IINFO_MIP_COEF_REDC_CUTS	Model	Number of coefficient reduction cuts generated.
LS_IINFO_MIP_GCD_CUTS	Model	Number of GCD cuts generated.
LS_IINFO_MIP_OBJ_CU	Model	Number of objective cuts generated.
LS_IINFO_MIP_BASIS_CUTS	Model	Number of basis cuts generated.
LS_IINFO_MIP_CARDGUB_CUTS	Model	Number of cardinality/GUB cuts generated.
LS_IINFO_MIP_CONTRA_CUTS	Model	Number of contra cuts generated.
LS_IINFO_MIP_CLIQUE_CUTS	Model	Number of clique cuts generated.
LS_IINFO_MIP_GUB_CONS	Model	Number of GUB constraints in the formulation.
LS_IINFO_MIP_GLB_CONS	Model	Number of GLB constraints in the formulation.
LS_IINFO_MIP_PLANTLOC_CONS	Model	Number of plant location constraints in the formulation.
LS_IINFO_MIP_DISAGG_CONS	Model	Number of disaggregation constraints in the formulation.
LS_IINFO_MIP_SB_CONS	Model	Number of single bound constraints in the formulation.
LS_IINFO_MIP_IKNAP_CONS	Model	Number of pure integer knapsack constraints in the formulation.
LS_IINFO_MIP_KNAP_CONS	Model	Number of knapsack constraints in the formulation.

LS_IINFO_MIP_NLP_CONS	Model	Number of nonlinear constraints in the formulation.
LS_IINFO_MIP_CONT_CONS	Model	Number of objective constraints in the formulation.
LS_DINFO_MIP_TOT_TIME	Model	Total MIP time including model I/O, optimization, heuristics.
LS_DINFO_MIP_OPT_TIME	Model	Total MIP optimization time.
LS_DINFO_MIP_HEU_TIME	Model	Total MIP heuristic time.
LS_IINFO_MIP_SOLSTATUS_LAST_BRANCH	Model	Solution status of the relaxation at the last branch.
LS_DINFO_MIP_SOLOBJVAL_LAST_BRANCH	Model	Objective value of the relaxation at the last branch.
LS_IINFO_MIP_HEU_LEVEL	Model	The current level for MIP heuristic engine.
LS_DINFO_MIP_PFEAS	Model	Primal infeasibility of the resident integer solution.
LS_DINFO_MIP_INTPFEAS	Model	Integer infeasibility of the resident integer solution.

Global Optimization Information

LS_DINFO_GOP_OBJ	Model	Objective value of the global optimal solution of a GOP.
LS_IINFO_GOP_SIM_ITER	Model	Number of simplex iterations performed for solving a GOP.
LS_IINFO_GOP_BAR_ITER	Model	Number of barrier iterations performed for solving a GOP.
LS_IINFO_GOP_NLP_ITER	Model	Number of NLP iterations performed for solving a GOP.
LS_DINFO_GOP_BESTBOUND	Model	Best bound on the objective value of a GOP.
LS_IINFO_GOP_STATUS	Model	Solution status of a GOP.
LS_IINFO_GOP_LPCOUNT	Model	Number of LPs solved for solving a GOP.
LS_IINFO_GOP_NLPCOUNT	Model	Number of NLPs solved for solving a GOP.
LS_IINFO_GOP_MIPCOUNT	Model	Number of MIPs solved for solving a GOP.
LS_IINFO_GOP_NEWSOL	Model	Whether a new GOP solution has been found or not.
LS_IINFO_GOP_BOX	Model	Number of explored boxes.
LS_IINFO_GOP_BBITER	Model	Number of iterations performed during a major GOP iteration.
LS_IINFO_GOP_SUBITER	Model	Number of iterations performed during a minor GOP iteration.
LS_IINFO_GOP_ACTIVEBOXES	Model	Number of active boxes at current state for solving a GOP.
LS_IINFO_GOP_TOT_TIME	Model	Total CPU time spent for solving a GOP.
LS_IINFO_GOP_MAXDEPTH	Model	Maximum depth of stack reached to store active boxes.
LS_IINFO_GOP_MIPBRANCH	Model	Number of branches created for solving a GOP.

Model Analysis Information

LS_IINFO_IIS_SIM_ITER	Model	Number of simplex iterations in IIS search.
LS_IINFO_IIS_BAR_ITER	Model	Number of barrier iterations in IIS search.
LS_IINFO_IIS_TOT_TIME	Model	Total CPU time spent for IIS search.
LS_IINFO_IIS_ACT_NODE	Model	Number of active sub problems remaining to complete the IIS search.
LS_IINFO_IIS_LPCOUNT	Model	Number of LPs solved during IIS search.
LS_IINFO_IIS_NLPCOUNT	Model	Number of NLPs solved during IIS search.
LS_IINFO_IIS_MIPCOUNT	Model	Number of MIPs solved during IIS search.
LS_IINFO_IUS_BAR_ITER	Model	Number of barrier iterations in IUS search.
LS_IINFO_IUS_SIM_ITER	Model	Number of simplex iterations in IUS search.
LS_IINFO_IUS_TOT_TIME	Model	Total CPU time spent for IIS search.
LS_IINFO_IUS_ACT_NODE	Model	Number of active sub problems remaining to complete the IUS search.
LS_IINFO_IUS_LPCOUNT	Model	Number of LPs solved during IUS search.
LS_IINFO_IUS_NLPCOUNT	Model	Number of NLPs solved during IUS search.
LS_IINFO_IUS_MIPCOUNT	Model	Number of MIPs solved during IUS search.

Stochastic Information

LS_DINFO_STOC_EVOBJ	Model	Expected value of the SP objective function, also called the Here-and-Now (HN) objective.
LS_DINFO_STOC_EVWS	Model	Expected value of the Wait-and-See (WS) model, which is a relaxation to the SP obtained by dropping the nonanticipativity restrictions.
LS_DINFO_STOC_EVPI	Model	Expected value of perfect information, which is defined as the difference between the expected value of the Wait-and-See objective value and the Here-and-Now objective function value.
LS_DINFO_STOC_EVAVR	Model	Optimal objective value of the restricted WS model where all stage-0 decisions are fixed at their respective values from the optimal solution of the Average-Model. The Average Model is the deterministic version of the original model constructed by replacing all random parameters with their expected values.
LS_DINFO_STOC_EVMU	Model	Expected value of modeling uncertainity, which is defined as the difference between the the Here-and-Now objective and the optimal value of the restricted-Wait-See objective. This value is also called the 'Value of Stochastic Solution'.
LS_DINFO_STOC_PINFEAS	Model	Primal infeasibility of the first stage solution.
LS_DINFO_STOC_DINFEAS	Model	Dual infeasibility of the first stage solution.
LS_DINFO_STOC_RELOPT_GAP	Model	Relative optimality gap at current solution.
LS_DINFO_STOC_ABSOPT_GAP	Model	Absolute optimality gap at current solution.
LS_IINFO_STOC_SIM_ITER	Model	Number of simplex iterations performed.
LS_IINFO_STOC_BAR_ITER	Model	Number of barrier iterations performed.
LS_IINFO_STOC_NLP_ITER	Model	Number of nonlinear iterations performed.

LS_IINFO_NUM_STOCPAR_RHS	Model	Number of stochastic parameters in the RHS.
LS_IINFO_NUM_STOCPAR_OBJ	Model	Number of stochastic parameters in the objective function.
LS_IINFO_NUM_STOCPAR_LB	Model	Number of stochastic parameters in the lower bound.
LS_IINFO_NUM_STOCPAR_UB	Model	Number of stochastic parameters in the upper bound.
LS_IINFO_NUM_STOCPAR_INSTR_OBJS	Model	Number of stochastic parameters in the instructions constituting the objective.
LS_IINFO_NUM_STOCPAR_INSTR_CONS	Model	Number of stochastic parameters in the instructions constituting the constraints.
LS_IINFO_NUM_STOCPAR_INSTR	Model	Total number of stochastic parameters in the instructions constituting the constraints and the objective.
LS_IINFO_NUM_STOCPAR_AIJ	Model	Number of stochastic parameters in the LP matrix.
LS_DINFO_STOC_TOTAL_TIME	Model	Total time elapsed in seconds to solve the model
LS_IINFO_STOC_STATUS	Model	Status of the SP model.
LS_IINFO_STOC_STAGE_BY_NODE	Model	Stage of the specified node.
LS_IINFO_STOC_NUM_SCENARIOS	Model	Number of scenarios (integer) in the scenario tree.
LS_DINFO_STOC_NUM_SCENARIOS	Model	Number of scenarios (double) in the scenario tree.
LS_IINFO_STOC_NUM_STAGES	Model	Number of stages in the model.
LS_IINFO_STOC_NUM_NODES	Model	Number of nodes in the scenario tree (integer).
LS_DINFO_STOC_NUM_NODES	Model	Number of nodes in the scenario tree (double).
LS_IINFO_STOC_NUM_NODES_STAGE	Model	Number of nodes that belong to specified stage in the scenario tree (integer).
LS_DINFO_STOC_NUM_NODES_STAGE	Model	Number of nodes that belong to specified stage in the scenario tree (double).
LS_IINFO_STOC_NUM_NODE_MODELS	Model	Number of node-models created or to be created.

LS_IINFO_STOC_NUM_COLS_BEFORE_NODE	Model	Column offset in DEQ of the first variable associated with the specified node.
LS_IINFO_STOC_NUM_ROWS_BEFORE_NODE	Model	Row offset in DEQ of the first variable associated with the specified node.
LS_IINFO_STOC_NUM_COLS_DETEQI	Model	Total number of columns in the implicit DEQ (integer).
LS_DINFO_STOC_NUM_COLS_DETEQI	Model	Total number of columns in the implicit DEQ (double).
LS_IINFO_STOC_NUM_ROWS_DETEQI	Model	Total number of rows in the implicit DEQ (integer).
LS_DINFO_STOC_NUM_ROWS_DETEQI	Model	Total number of rows in the implicit DEQ (double).
LS_IINFO_STOC_NUM_COLS_DETEQE	Model	Total number of columns in the explict DEQ (integer).
LS_DINFO_STOC_NUM_COLS_DETEQE	Model	Total number of columns in the explict DEQ (double).
LS_IINFO_STOC_NUM_ROWS_DETEQE	Model	Total number of rows in the explict DEQ (integer).
LS_DINFO_STOC_NUM_ROWS_DETEQE	Model	Total number of rows in the explict DEQ (double).
LS_IINFO_STOC_NUM_COLS_NAC	Model	Total number of columns in non-anticipativity block.
LS_IINFO_STOC_NUM_ROWS_NAC	Model	Total number of rows in non-anticipativity block.
LS_IINFO_STOC_NUM_COLS_CORE	Model	Total number of columns in core model.
LS_IINFO_STOC_NUM_ROWS_CORE	Model	Total number of rows in core model.
LS_IINFO_STOC_NUM_COLS_STAGE	Model	Total number of columns in core model in the specified stage.
LS_IINFO_STOC_NUM_ROWS_STAGE	Model	Total number of rows in core model in the specified stage.
LS_IINFO_STOC_NUM_BENDERS_FCUTS	Model	Total number of feasibility cuts generated during NBD iterations.
LS_IINFO_STOC_NUM_BENDERS_OCUTS	Model	Total number of optimality cuts generated during NBD iterations.
LS_IINFO_DIST_TYPE	Model	Distribution type of the sample
LS_IINFO_SAMP_SIZE	Model	Sample size.

LS_DINFO_SAMP_MEAN	Model	Sample mean.
LS_DINFO_SAMP_STD	Model	Sample standard deviation.
LS_DINFO_SAMP_SKEWNESS	Model	Sample skewness.
LS_DINFO_SAMP_KURTOSIS	Model	Sample kurtosis.
LS_IINFO_STOC_NUM_QCP_CONS_DETEQE	Model	Total number of quadratic constraints in the explicit deterministic equivalent.
LS_IINFO_STOC_NUM_CONT_CONS_DETEQE	Model	Total number of continuous constraints in the explicit deterministic equivalent.
LS_IINFO_STOC_NUM_INT_CONS_DETEQE	Model	Total number of constraints with general integer variables in the explicit deterministic equivalent.
LS_IINFO_STOC_NUM_BIN_CONS_DETEQE	Model	Total number of constraints with binary variables in the explicit deterministic equivalent.
LS_IINFO_STOC_NUM_QCP_VARS_DETEQE	Model	Total number of quadratic variables in the explicit deterministic equivalent.
LS_IINFO_STOC_NUM_NONZ_DETEQE	Model	Total number of nonzeros in the explicit deterministic equivalent.
LS_IINFO_STOC_NUM_BIN_DETEQE	Model	Total number of binaries in the explicit deterministic equivalent.
LS_IINFO_STOC_NUM_INT_DETEQE	Model	Total number of general integer variables in the explicit deterministic equivalent.
LS_IINFO_STOC_NUM_CONT_DETEQE	Model	Total number of continuous variables in the explicit deterministic equivalent.
LS_IINFO_STOC_NUM_QC_NONZ_DETEQE	Model	Total number of quadratic nonzeros in the explicit deterministic equivalent.
LS_IINFO_STOC_NUM_NLP_NONZ_DETEQE	Model	Total number of nonlinear nonzeros in the constraints of explicit deterministic equivalent.
LS_IINFO_STOC_NUM_NLPOBJ_NONZ_DETEQE	Model	Total number of nonlinear nonzeros in the objective function of explicit deterministic equivalent.
LS_IINFO_STOC_NUM_QCP_CONS_DETEQI	Model	Total number of quadratic constraints in the implicit deterministic equivalent.
LS_IINFO_STOC_NUM_CONT_CONS_DETEQI	Model	Total number of continuous constraints in the implicit deterministic equivalent.

LS_IINFO_STOC_NUM_INT_CONS_DETEQI	Model	Total number of constraints with general integer variables in the implicit deterministic equivalent.
LS_IINFO_STOC_NUM_BIN_CONS_DETEQI	Model	Total number of constraints with binary variables in the implicit deterministic equivalent.
LS_IINFO_STOC_NUM_QCP_VARS_DETEQI	Model	Total number of quadratic variables in the implicit deterministic equivalent.
LS_IINFO_STOC_NUM_NONZ_DETEQI	Model	Total number of nonzeros in the implicit deterministic equivalent.
LS_IINFO_STOC_NUM_BIN_DETEQI	Model	Total number of binaries in the implicit deterministic equivalent.
LS_IINFO_STOC_NUM_INT_DETEQI	Model	Total number of general integer variables in the implicit deterministic equivalent.
LS_IINFO_STOC_NUM_CONT_DETEQI	Model	Total number of continuous variables in the implicit deterministic equivalent.
LS_IINFO_STOC_NUM_QC_NONZ_DETEQI	Model	Total number of quadratic nonzeros in the implicit deterministic equivalent.
LS_IINFO_STOC_NUM_NLP_NONZ_DETEQI	Model	Total number of nonlinear nonzeros in the constraints of implicit deterministic equivalent.
LS_IINFO_STOC_NUM_NLPOBJ_NONZ_DETEQI	Model	Total number of nonlinear nonzeros in the objective function of implicit deterministic equivalent.
LS_IINFO_STOC_NUM_EVENTS_BLOCK	Model	Total number of block events.
LS_IINFO_STOC_NUM_EVENTS_DISCRETE	Model	Total number of independent events with a discrete distribution.
LS_IINFO_STOC_NUM_EVENTS_PARAMETRIC	Model	Total number of independent events with a parametric distribution.
LS_IINFO_STOC_NUM_EVENTS_SCENARIOS	Model	Total number of events loaded explictly as a scenario.
LS_IINFO_STOC_PARENT_NODE	Model	Index of a node's parent.
LS_IINFO_STOC_ELDEST_CHILD_NODE	Model	Index of a node's eldest child.
LS_IINFO_STOC_NUM_CHILD_NODES	Model	Total number of childs a node has.
LS_IINFO_INFORUNB_SCEN_IDX	Model	Index of the infeasible or unbounded scenario.
LS_IINFO_DIST_NARG	Model	Number of arguments of a distribution sample.

LS_IINFO_SAMP_VARCONTROL_METHOD	Model	Variance reduction/control method used in generating the sample.
LS_IINFO_STOC_NUM_NLP_VARS_DETEQE	Model	Total number of nonlinear variables in the explicit deterministic equivalent.
LS_IINFO_STOC_NUM_NLP_CONS_DETEQE	Model	Total number of nonlinear constraints in the explicit deterministic equivalent.
LS_DINFO_STOC_EVOBJ_LB	Model	Best lower bound on expected value of the objective function.
LS_DINFO_STOC_EVOBJ_UB	Model	Best upper bound on expected value of the objective function.
LS_DINFO_STOC_AVROBJ	Model	Expected value of average model's objective.
LS_DINFO_SAMP_MEDIAN	Model	Sample median.
LS_DINFO_DIST_MEDIAN	Model	Distribution (population) median.

Model Loading Routines

The routines described in this section allow you to pass a model to LINDO API directly through memory. LINDO API expects the formulation to be in sparse format. In other words, only nonzero coefficients are passed. For details on sparse representation, see the section titled *Sparse Matrix Representation* in Chapter 1, *Introduction*. Before using routines described in this section, be aware that another way of passing a model to the LINDO API is by using one of the LSreadLINDOFile, LSreadMPSFile, and LSreadMPIFile routines described earlier in this chapter. In fact, for debugging reasons, you may want to consider passing your model to the LINDO API by file using the LSreadXXFile routines rather than with the direct memory methods described below. If a model is not behaving as you think it should, it is relatively easy to send a file to the Tech support people at LINDO. If you are confident that your formulation is debugged, and you need high performance, or the ability to run several models simultaneously, as in a web-based application, then you can always switch to the direct memory transfer routines described below.

Note: LINDO API keeps its own copies of the data passed via the input arguments in the model space. Therefore, the user can free the local copies after the call completes successfully.

LSloadConeData()

Description:

Loads quadratic cone data into a model structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadConeData (pLSmodel pModel, int nCone, char *pszConeTypes, int *paiConebegcone, int *paiConecols)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the problem data.
nCone	Number of cones to add.
pszConeTypes	A pointer to a character vector containing the type of each cone being added. Valid values for each cone are 'Q' and 'R'. The length of this vector is equal to *nCone*.
paiConebegcone	A pointer to an integer vector containing the index of the first variable that appears in the definition of each cone. This vector must have nCone+1 entries. The last entry will be the index of the next appended cone, assuming one was to be appended. If paiConebegcone [i] < paiConebegcone [i-1], then LSERR_ERROR_IN_INPUT is returned.
paiConecols	A pointer to an integer vector containing the indices of variables representing each cone. The length of this vector is equal to paiConebegcone[nCone].

LSloadInstruct()

Description:

Loads instruction lists into a model structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadInstruct (pLSmodel pModel, int nCons, int nObjs, int nVars, int nNums, int *panObjSense, char *pacConType, char *pacVarType, int *panCode, int nCode, int *paiVars, double *padVals, double *padX0, int *paiObj, int *panObj, int *paiRows, int *panRows, double *padL, double *padU)

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.
nCons	Number of constraints in the model.
nObjs	Number of objectives in the model. Currently, there is only support for a single objective (i.e., nObjs = 1).
nVars	Number of variables in the model.
nNums	Number of real numbers in the model.
panObjSense	A pointer to an integer vector containing the indicator stating whether the objective is to be maximized or minimized. Valid

	values are LS_MAX or LS_MIN, respectively. The length of this vector is equal to *nObjs*. Currently, there is only support for a single objective.
pacConType	A pointer to a character vector containing the type of each constraint. Each constraint is represented by a single byte in the array. Valid values for each constraint are 'L', 'E', 'G', or 'N' for less-than-or-equal-to, equal to, great-than-or-equal-to, or neutral, respectively. The length of this vector is equal to *nCons*.
pacVarType	A pointer to a character vector containing the type of each variable. Valid values for each variable are 'C', 'B', or 'I', for continuous, binary, or general integer, respectively. The length of this vector is equal to *nVars*. This value may be NULL on input, in which case all variables will be assumed to be continuous.
panCode	A pointer to an integer vector containing the instruction list. The length of this vector is equal to *nCode*. For details on instruction list representation, see the section titled *Instruction-List Style Interface* in Chapter 7, *Solving Nonlinear Programs*.
nCode	Number of items in the instruction list.
paiVars	A pointer to an integer vector containing the variable index. The length of this vector is equal to *nVars*. This pointer may be set to NULL if the variable index is consistent with the variable position in the variable array.
padVals	A pointer to a double precision vector containing the value of each real number in the model. The length of this vector is equal to *nNums*.
padX0	A pointer to a double precision vector containing starting values for each variable in the given model. The length of this vector is equal to *nVars*.
paiObj	A pointer to an integer vector containing the beginning positions on the instruction list for each objective row. The length of this vector is equal to *nObjs*. Currently, there is only support for a single objective.
panObj	A pointer to an integer vector containing the length of instruction code (i.e., the number of individual instruction items) for each objective row. The length of this vector is equal to *nObjs*. Currently, there is only support for a single objective.
paiRows	A pointer to an integer vector containing the beginning positions on the instruction list for each constraint row. The length of this vector is equal to *nCons*.
panRows	A pointer to an integer vector containing the length of instruction code (i.e., the number of individual instruction

	items) for each constraint row. The length of this vector is equal to *nCons*.
padL	A pointer to a double precision vector containing the lower bound of each variable. If there is no lower bound on the variable, then this value should be set to –LS_INFINITY. If padL is NULL, then the lower bounds are internally set to zero.
padU	A pointer to a double precision vector containing the upper bound of each variable. If there is no upper bound on the variable, then this value should be set to LS_INFINITY. If padU is NULL, then the upper bounds are internally set to LS_INFINITY.

Remarks:

- The instruction lists for the objective and constraints are all carried by the same code vector, *panCode, to load into LINDO API model structure.
- The index vector *paiVars can be used to store the user-specified variable index. Currently, the values supplied in paiVars[ ] are unimportant.

LSloadLPData()

Description: v

Loads the given LP data into the *LSmodel* data structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadLPData (pLSmodel pModel, int nCons, int nVars, int dObjsense, double dObjconst, double *padC, double *padB, char *pachContypes, int nAnnz, int *paiAcols, int *pacAcols, double *padAcoef, int *paiArows, double *padL, double *padU)

Arguments:

Name	**Description**
pModel	A pointer to an instance of *LSmodel* in which to place the problem data.
nCons	Number of constraints in the model.
nVars	Number of variables in the model.
dObjsense	An indicator stating whether the objective is to be maximized or minimized. Valid values are LS_MAX or LS_MIN.
dObjconst	A constant value to be added to the objective value.
padC	A pointer to a double precision vector containing the objective coefficients.

padB	A pointer to a double precision vector containing the constraint right-hand side coefficients.
pachContypes	A pointer to a character vector containing the type of each constraint. Each constraint is represented by a single byte in the array. Valid values for each constraint are 'L', 'E', 'G', or 'N' for less-than-or-equal-to, equal-to, greater-than-or-equal-to, or neutral, respectively.
nAnnz	The number of nonzeros in the constraint matrix.
paiAcols	A pointer to an integer vector containing the index of the first nonzero in each column. This vector must have nVars+1 entries. The last entry will be the index of the next appended column, assuming one was to be appended. If paiAcols[i] < paiAcols[i-1], then LSERR_ERROR_IN_INPUT is returned.
pacAcols	A pointer to an integer vector containing the length of each column. Note that the length of a column can be set to be smaller than the values paiAcols would suggest (i.e., it is possible for pacAcols[*i*] < paiAcols[*i*+1] – paiAcols[*i*]). This may be desirable in order to prevent memory reallocations in the event that any rows are added to the model. If the columns are packed tight (i.e., the length of a column *i* is equal to paiAcols[*i*+1] – paiAcols[*i*] for all *i*), then pacAcols can be set to NULL on input.
padAcoef	A pointer to a double precision vector containing the nonzero coefficients of the constraint matrix.
paiArows	A pointer to an integer vector containing the row indices of the nonzeros in the constraint matrix. If any row index is not in the range [0, nCons -1], LSERR_INDEX_OUT_OF_RANGE is returned.
padL	A pointer to a double precision vector containing the lower bound of each variable. If there is no lower bound on the variable, then this value should be set to -LS_INFINITY. If it is NULL, then the lower bounds are internally set to zero.
padU	A pointer to a double precision vector containing the upper bound of each variable. If there is no upper bound on the variable, then this value should be set to LS_INFINITY. If it is NULL, then the upper bounds are internally set to LS_INFINITY.

Remarks:

- The data from each of the arrays passed to this routine are actually copied into arrays within the *LSmodel* structure. Therefore, the calling routine can free the memory if the information is no longer needed.
- To retrieve the LP's data from the model structure, see routine *LSgetLPData*().

LSloadNameData()

Description:

Loads the given name data (e.g., row and column names), into the *LSmodel* data structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadNameData(pLSmodel pModel, char *pszTitle, char *pszObjName, char *pszRhsName, char *pszRngName, char *pszBndname, char **paszConNames, char **paszVarNames, char **paszConeNames)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the problem data.
pszTitle	A pointer to a null terminated string containing the title of the problem.
pszObjName	A pointer to a null terminated string containing the name of the objective.
pszRhsName	A pointer to a null terminated string containing the name of the right-hand side vector.
pszRngName	A pointer to a null terminated string containing the name of the range vector.
pszBndname	A pointer to a null terminated string containing the name of the bounds vector.
paszConNames	A pointer to an array of pointers to the null terminated constraint names.
paszVarNames	A pointer to an array of pointers to the null terminated variable names.
paszConeNames	A pointer to an array of pointers to the null terminated cone names.

Remarks:

- The data from each of the arrays passed to this routine are actually copied into arrays within the *LSmodel* structure. Therefore, the calling routine can free the memory if the information is no longer needed.
- Any of the pointers to name data passed to this routine may be set to NULL if the information is not relevant.

LSloadNLPData()

Description:

Loads a nonlinear program's data into the model data structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadNLPData (pLSmodel pModel, int * paiCols, int * pacCols, double * padCoef, int * paiRows, int nObj, int *paiObj, double *padObjCoef)

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.
paiCols	A pointer to an integer vector containing the index of the first nonlinear nonzero in each column. This vector must have *nVars*+1 entries, where *nVars* is the number of variables. The last entry will be the index of the next appended column, assuming one was to be appended.
pacCols	A pointer to an integer vector containing the number of nonlinear elements in each column.
padCoef	A pointer to a double precision vector containing initial values of the nonzero coefficients in the (Jacobian) matrix. It may be set to NULL, in which case, LINDO API will compute an initial matrix.
paiRows	A pointer to an integer vector containing the row indices of the nonlinear elements.
nObj	An integer containing the number of nonlinear variables in the objective.
paiObj	A pointer to an integer vector containing the column indices of nonlinear variables in the objective function.
padObjCoef	A pointer to double precision vector containing the initial nonzero coefficients in the objective. It may be set to NULL, in which case, LINDO API will compute an initial gradient vector.

Remarks:

- Currently, the values supplied in *padCoef* are unimportant and can always be set to NULL.
- Note, a nonzero constraint matrix must be established before calling *LSloadNLPData*(). This is accomplished through a call to *LSloadLPData*(). The subsequent call to *LSloadNLPData*() simply identifies the nonzeros in the matrix that are nonlinear (i.e., not constant). As an example, consider the nonlinear row: $3x + y^2 - 1 \leq 0$. In this row, *x* appears linearly and, therefore, has a fixed coefficient of value 3. The variable *y,* on the other hand, appears nonlinearly and does not have a fixed coefficient. Its coefficient at any given point must be determined through finite differences or a call to *pGradcalc*(). Note that a variable appearing in both linear and nonlinear terms should be treated nonlinearly and has no fixed coefficient (e.g., x + x^2). Identifying the fixed coefficients allows LINDO API to minimize the amount of work required to compute gradients.

LSloadQCData()

Description:

Loads quadratic program data into the *LSmodel* data structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadQCData(pLSmodel pModel, int nQCnnz, int *paiQCrows, int *paiQCcols1, int *paiQCcols2, double *padQCcoef)

Arguments:

Name	Description
pModel	An instance of *LSmodel* in which to place the problem data.
nQCnnz	The total number of nonzeros in quadratic coefficient matrices.
paiQCrows	A pointer to a vector containing the index of the constraint associated with each nonzero quadratic term. This vector must have *nQCnnz* entries.
paiQCcols1	A pointer to a vector containing the index of the first variable defining each quadratic term. This vector must have *nQCnnz* entries.
paiQCcols2	A pointer to a vector containing the index of the second variable defining each quadratic term. This vector must have *nQCnnz* entries.
padQCcoef	A pointer to a vector containing the nonzero coefficients in the quadratic matrix. This vector must also have *nQCnnz* entries.

Remarks:

- The data from each of the arrays passed to this routine are actually copied into arrays within the *LSmodel* structure. Therefore, the calling routine can free the memory if the information is no longer needed.
- The quadratic matrices are assumed to be symmetric.
- Only the upper triangular part of the quadratic matrices must be specified.
- For variations on the above, e.g. if a matrix is not naturally symmetric, see Chapter 5, *Solving Quadratic Programs*, for more information.

LSloadSemiContData()

Description:

Loads semi-continuous data into the *Lsmodel* data structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadSemiContData(pLSmodel pModel, int nSC, int *piVarndx, double *padl, double *padu)

Arguments:

Name	Description
pModel	An instance of *LSmodel* in which to place the problem data.
nSC	The number of semi-continuous variables.
piVarndx	A pointer to a vector containing the indices of semi-continuous variables. This vector must have *nSC* entries.
padl	A pointer to a vector containing the lower bound associated with each semi-continuous variable. This vector must also have *nSC* entries.
padu	A pointer to a vector containing the upper bound associated with each semi-continuous variable. This vector must also have *nSC* entries.

LSloadSETSData()

Description:

Loads special sets data into the *Lsmodel* data structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadSETSData(pLSmodel pModel, int nSETS, char *pszSETStype, int *paiCARDnum, int *paiSETSbegcol, int *paiSETScols)

Arguments:

Name	Description
pModel	An instance of *LSmodel* in which to place the problem data.
nSETS	Number of sets to load.
pszSETStype	A pointer to a character vector containing the type of each set. Valid values for each set are : LS_MIP_SET_CARD LS_MIP_SET_SOS1 LS_MIP_SET_SOS2 LS_MIP_SET_SOS3
paiCARDnum	A pointer to an integer vector containing set cardinalities. This vector must have *nSETS* entries. The set cardinalities are taken into account only for sets with pszSETStype[i] = LS_MIP_SET_CARD.
paiSETSbegcol	A pointer to an integer vector containing the index of the first variable in each set. This vector must have nSETS+1 entries. The last entry will be the index of the next appended set, assuming one was to be appended. If paiSETSbegcol[i] < paiSETSbegcol [i-1], then LSERR_ERROR_IN_INPUT is returned.
paiSETScols	A pointer to an integer vector containing the indices of variables in each set. If any index is not in the range [0, nVars -1], LSERR_INDEX_OUT_OF_RANGE is returned.

LSloadVarType()

Description:

Loads the given MIP (mixed-integer program) data into the *LSmodel* data structure. The old name for this function is *LSloadMIPData*().

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadVarType(pLSmodel pModel, char *pachVartypes)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the MIP data.
pachVartypes	A pointer to a character vector containing the type of each variable. Valid values for each variable are 'C', 'B', or 'I' for continuous, binary, or general integer, respectively. This value may be NULL on input, in which case all variables will be assumed to be continuous.

Remarks:

- The ability to solve mixed-integer programs is an optional feature. Not all installations will have this capability. To determine if your license includes MIP functionality, use *LSgetModelIntParameter*() with license information access macros.
- The data from each of the arrays passed to this routine are actually copied into arrays within the *LSmodel* structure. Therefore, the calling routine can free the memory if the information is no longer needed.
- *LSloadLPData*() must be called prior to calling this routine.
- *LScreateModel*() must be called prior to calling this routine.
- To load variable branching priorities, see *LSloadVarPriorities*().
- *LSloadLPData* must have been called previously.

LSloadStringData()

Description:

Loads a vector of strings into the *LSmodel* data structure and gets sort order.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadStringData(pLSmodel pModel, int nStrings, char **paszStrings)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the string data.
nStrings	Number of strings to load
PaszStrings	A pointer to an array of pointers to the null terminated strings..

Remarks:

- All strings to be used in a model need to be loaded into LINDO API with either a sequence of calls to LSloadString followed by a final call to LSbuildStringData, or a single call to LSloadStringData. These calls must be made before strings can be referred to in the instruction list through the operators EP_PUSH_STR or EP_VPUSH_STR. The vector of strings loaded is automatically sorted by finalizing the loading with a call to *LSbuildStringData*. An index, starting from 1, is assigned to each unique string and this index can be used to access the string values by a call to *LSgetStringValue*.

LSloadString()

Description:

Load a single string into the *LSmodel* data structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadString(pLSmodel pModel, char *szString)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the string data.
szString	A pointer to a null terminated string .

Remarks:

- See also LSbuildStringData, and LSloadStringData.

LSbuildStringData()

Description:

Gets sort order of all strings loaded by previous calls to LSloadString, and assigns a unique value to each unique string.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSbuildStringData(pLSmodel pModel)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the string data.

Remarks:

See also, LSloadString and LSloadStringData.

LSdeleteStringData()

Description:

Delete the string values data

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSdeleteStringData(pLSmodel pModel)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the string data.

Remarks:

- Please refer to LSloadStringData for the detailed string support.

LSdeleteString()

Description:

Delete the complete string data, including the string vector and values.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSdeleteString(pLSmodel pModel)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the string data.

Remarks:

- Please refer to LSloadStringData for the detailed string support.

LSgetStringValue()

Description:

Retrieve a string value for a specified string index.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetStringValue(pLSmodel pModel, int nStringIdx, double pdStrinVal)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* in which to place the string data.
nStringIdx	An integer containing the index of the string whose value you wish to retrieve.
pdStrinVal	A pointer to a double precision quantity that returns the string value.

Remarks:

- Please refer to LSloadStringData for the detailed string support.

LSloadSampleSizes ()

Description:

Loads sample sizes per stage for the stochastic model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSloadSampleSizes (pLSmodel pModel, int * panSampleSize)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
panSampleSize	An integer vector specifying the stage sample sizes. The length of this vector should be at least the number of stages in the model.

LSsetNumStages ()

Description:

Set number of stages in the model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsetNumStages (pLSmodel pModel, int numStages)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
numStages	An integer specifying the number of stages in the model.

LSloadConstraintStages ()

Description:

Load stage structure of the constraints in the model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSloadConstraintStages (pLSmodel pModel, int * panRstage)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
panRstage	A vector in which information about the stage membership of the constraints is held. The length of this vector is equal to the number of constraints in the model. If constraint `i` belongs to stage `k` , then `panRstage[i] = k-1`

LSloadVariableStages ()

Description:

Load stage structure of the variables in the model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSloadVariableStages (pLSmodel pModel, int * panCstage)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
panCstage	A vector in which information about the stage membership of the variables is held. The length of this vector is equal to the number of variables in the model. If variable i belongs to stage k , then panCstage[i] = k-1

LSloadStocParData ()

Description:

Load stage structure of the stochastic parameters (SPARs) in the model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSloadStocParData (pLSmodel pModel, int * panSvarStage, double * padSvarValue)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
panSvarStage	An integer vector specifying the stages of SPARs. The length of this vector is equal to the number of SPARs in the model. if SPAR i belongs to stage k , then panSvarStage[i] = k-1
padSvarValue	A double vector specifying the default values of SPARs. The length of this vector is equal to the number of SPARs in the model. If NULL, a value of zero is assumed for all SPARS.

Remarks:

- Length of SPARS can be retrieved with LS_IINFO_NUM_SPARS macro.

LSaddDiscreteIndep ()

Description:

Adds a new discrete independent stochastic parameter to the SP model. The positions of stochastic parameters are specified with either `(iRow,jCol)` or `iStv` , but not with both. For SP models where core model is described with an instruction list, `iStv` have to be used.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSaddDiscreteIndep (pLSmodel pModel, int iRow, int jCol, int iStv, int nRealizations, double * padProbs, double * padVals, int iModifyRule)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iRow	An integer specifying the row index of the stochastic parameter. It should be ignored if iStv will be specified.
jCol	An integer specifying the column index of the stochastic parameter. It should be ignored if iStv will be specified.
iStv	An integer specifying the index of stochastic parameter in the instruction list. It should ignored if (iRow,jCol) is specified.
nRealizations	An integer specifying the number of all possible realizations for the specified stochastic parameter.
padProbs	A double vector of probabilities associated with the realizations of the stochastic parameter. The length of this vector should be nRealizations or more.
padVals	A double vector of values associated with the probabilities. The length of this vector should be nRealizations or more.
iModifyRule	A flag indicating whether stochastic parameters update the core model by adding or replacing.

LSaddParamDistIndep ()

Description:

Adds a new independent stochastic parameter with a parameteric distribution to the SP model. The positions of stochastic parameters are specified with either `(iRow, jCol)` or `iStv`, but not with both. For SP models where core model is described with an instruction list, `iStv` have to be used.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSaddParamDistIndep (pLSmodel pModel, int iRow, int jCol, int iStv, int nDistType, int nParams, double * padParams, int iModifyRule)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iRow	An integer specifying the row index of the stochastic parameter. It should be ignored if iStv will be specified.
jCol	An integer specifying the column index of the stochastic parameter. It should be ignored if iStv will be specified.
iStv	An integer specifying the index of stochastic parameter in the instruction list. It should ignored if (iRow, jCol) is specified.
nDistType	An integer specifying the parametric distribution type. See the 'Distributions' table for possible values.
padParams	An double vector specifying the parameters of given distribution.
nParams	An integer specifying the length of padParams .
iModifyRule	A flag indicating whether stochastic parameters update the core model by adding or replacing. Possible values are: • LS_REPLACE • LS_ADD

LSaddDiscreteBlocks ()

Description:

Adds a new discrete stochastic block to the SP model. The positions of stochastic parameters are specified with either `(paiArows,paiAcols)` or `paiStvs` , but not with both. For SP models where core model is described with an instruction list, `paiStvs` have to be used.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSaddDiscreteBlocks (pLSmodel pModel, int iStage, int nBlockEvents, double * padProb, int * pakEvent, int * paiArows, int * paiAcols, int * paiStvs, double * padVals, int iModifyRule)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iStage	An integer specifying the stage of the stochastic block.
nBlockEvents	An integer specifying the number of discrete events in the block.
padProb	An double vector of event probabilities.
pakEvent	An integer vector of starting positions of events in the sparse matrix or instruction list. This vector should have nBlockEvents+1 elements or more.
paiArows	An integer vector of row indices of stochastic parameters. This vector should have pakEvent[nBlockEvents] elements. It should be NULL when paiStvs is specified.
paiAcols	An integer vector of column indices of stochastic parameters. This vector should have pakEvent[nBlockEvents] elements. It should be NULL when paiStvs is specified.
paiStvs	An integer vector of indices of stochastic parameters in the instruction list. The length of this vector should be pakEvent[nBlockEvents] or more. It should be NULL when (paiArows,paiAcols) is specified.
padVals	A double vector of stochastic values associated with the stochastic parameters listed in paiStvs or (paiArows,paiAcols) The length of this vector should be pakEvent[nBlockEvents] or more.
iModifyRule	A flag indicating whether stochastic parameters update the core model by adding or replacing.

LSaddScenario ()

Description:

Adds a new scenario block to the SP model. The positions of the stochastic parameters are specified with either `(paiArows,paiAcols)` or `paiStvs` , but not with both.

For SP models where core model is described with an instruction list, `paiStvs` have to be used.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSaddScenario (pLSmodel pModel, int jScenario, int iParentScen, int iStage, double dProb, int nElems, int * paiArows, int * paiAcols, int * paiStvs, double * padVals, int iModifyRule)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
jScenario	An integer specifying the index of the new scenario to be added.
iParentScen	Index of the parent scenario.
iStage	Index of the stage the new scenario branches from its parent.
dProb	A double scalar specifying the scenario probability.
nElems	The number of stochastic parameters realized at stage iStage in the new scenario.
paiArows	An integer vector of the row indices of stochastic parameters. This vector should have nElems elements or more. It should be NULL when paiStvs is specified.
paiAcols	An integer vector of the column indices of stochastic parameters. This vector should have nElems elements or more. It should be NULL when paiStvs is specified.
paiStvs	An integer vector of indices of stochastic parameters in instruction list. This vector should have nElems elements or more. It should be NULL when (paiArows,paiAcols) is specified.
padVals	A double vector of values of stochastic parameters. This vector should have nElems elements or more.
iModifyRule	A flag indicating whether stochastic parameters update the core model by adding or replacing.

LSloadStocParNames ()

Description:

This routine loads name data for stochastic parameters into the specified LSmodel structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSloadStocParNames (pLSmodel pModel, int numVars, char ** stv_names)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
numVars	An integer specifying the number of stochastic parameters.
stv_names	An array of pointers to the stochastic parameter names. This value can be NULL.

Remarks:

The data from each of the arrays passed to this routine are actually copied into arrays within the LSmodel structure. Therefore, the calling routine can free the memory if the information is no longer needed.

Solver Initialization Routines

The routines in this section allow you to pass the internal solver starting-point information when solving linear models and branching priorities when solving mixed-integer models.

LSloadBasis()

Description:

Provides a starting basis for the simplex method. A starting basis is frequently referred to as being a "warm start".

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadBasis(pLSmodel pModel, int *panCstatus, int *panRstatus)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* containing the model for which you are providing the basis.
panCstatus	A pointer to an integer vector containing the status of each column in the given model. The length of this vector is equal to the number of variables in the model. The *i*-th element of the array corresponds to the *i*-th variable in the model. Set each variable's element to 0, −1, −2, or −3 for Basic, Nonbasic at lower bound, Nonbasic at upper bound, or Free and nonbasic at zero value, respectively.
panRstatus	A pointer to an integer vector in which information about the status of the rows is to be placed. The length of this vector is equal to the number of constraints in the model. The *i*-th element of the array corresponds to the *i*-th row in the model.

	Set each row's element to 0 or −1 if row's associated slack variable is basic or row's associated slack variable is nonbasic at zero, respectively.

Remarks:

- To retrieve a basis use LSgetBasis().
- *LSloadBasis()* does not require the row indices that the variables are basic in. Setting all basic variables to a nonnegative integer is sufficient to specify a basis.
- *LSgetBasis()*, in addition to the indices of basic variables, returns also the indices of the rows that variables are basic in.

LSloadVarPriorities()

Description:

Provides priorities for each variable for use in branch-and-bound.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadVarPriorities(pLSmodel pModel, int *panCprior)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
panCprior	A pointer to a vector containing the priority of each column in the given model. The length of this vector is equal to the number of variables in the model. A valid priority value is any nonnegative integer value. Variables with higher priority values are given higher branching priority.

Remarks:

- Although this routine requires priorities for all variables, the branch-and-bound solver only makes use of the priorities on the integer variables. The priorities on continuous variables are ignored.
- To read priorities from a disk file, see *LSreadVarPriorities*().

LSloadVarStartPoint()

Description:

Provides an initial starting point for nonlinear and branch-and-bound solvers.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadVarStartPoint(pLSmodel pModel, double *padPrimal)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
padPrimal	A pointer to a double precision vector containing starting values for each variable in the given model. The length of this vector is equal to the number of variables in the model.

Remarks:

- The nonlinear solver may modify the initial solution to improve its quality if sequential linear programming (SLP) step directions are allowed.
- Although this routine requires values for all variables, the branch-and-bound solver will only make use of the values for the integer variables.

LSloadBlockStructure()

Description:

Provides a block structure for the constraint matrix by specifying block memberships of each variable and constraint.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSloadBlockStructure(pLSmodel pModel, int nBlock, int *panRblock, int *panCblock, int nType)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
nBlock	An integer scalar that contains the number of blocks the model matrix decomposes into.
panRblock	A pointer to an integer vector in which information about the block membership of the constraints is placed. The length of this vector must be ≥ the number of constraints in the model. The *i*-th element of this array keeps the information on the *i*-th constraint as follows: 0: The row is a member of the linking (row) block. *k*>0: The row is a member of the *k*-th block. where 1 <= *k* <= *nBlock*.
panCblock	A pointer to an integer vector in which information about the block membership of the variables is placed. The length of this vector must be ≥ the number of variables in the model. The *j*-th

	element of this array contains information on the *j*-th column as follows: 0: The column is a member of the linking (column) block. *k*>0: The column is a member of the *k*-th block. where 1 <= *k* <= *nBlock*.
nType	An integer scalar indicating the type of decomposition loaded. The possible values are identified with the following macros: • LS_LINK_BLOCKS_COLS: The decomposed model has dual angular structure (linking columns). • LS_LINK_BLOCKS_ROWS: The decomposed model has block angular structure (linking rows). • LS_LINK_BLOCKS_BOTH: The decomposed model has both dual and block angular structure (linking rows and columns)

Remarks:

- For more information on decomposition and linking structures, refer to Chapter 10, *Analyzing Models and Solutions*.
- See also *LSfindBlockStructure()*.

LSreadVarPriorities()

Description:

Provides branching priorities for integer variables from a disk file.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSreadVarPriorities(pLSmodel pModel, char *pszFname)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
pszFname	A pointer to a null terminated string containing the name of the file from which to read the priorities.

Remarks:

- This routine expects one variable name and one integer priority value per record. The variable name must appear first followed by a nonnegative integer priority value. You need not specify priorities on all variables. If desired, you may specify priorities on only a subset of the variables.
- To pass priorities directly through an array, see *LSloadVarPriorities*().

LSreadVarStartPoint()

Description:

Provides initial values for variables from a file.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSreadVarStartPoint(pLSmodel pModel, char *pszFname)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
pszFname	A pointer to a null terminated string containing the name of the file from which to read the starting values.

Remarks:

- This routine expects one variable name and one value per record. The variable name must appear first followed by a starting value. To pass initial values directly through an array, see *LSloadVarStartPoint*().

Optimization Routines

The routines in this section are called to invoke LINDO API's solver. There are three routines—*LSsolveMIP*(), *LSoptimize*(), and *LSsolveGOP*(). *LSsolveMIP*() should be called when the model has one or more integer variables, while *LSoptimize*() should be called when all the variables are continuous. *LSsolveGOP*() should be called for global optimization of nonlinear models.

LSoptimize()

Description:

Optimizes a continuous model by a given method.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSoptimize(pLSmodel pModel, int nMethod, int *pnStatus)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nMethod	A parameter indicating the solver to be used in optimizing the problem. Current options for this parameter are • LS_METHOD_FREE: 0, • LS_METHOD_PSIMPLEX: 1,

	• LS_METHOD_DSIMPLEX: 2, • LS_METHOD_BARRIER: 3, • LS_METHOD_NLP: 4. When the method is set to LS_METHOD_FREE, LINDO API will decide the best solver to use. The remaining four methods correspond to the primal simplex, dual simplex, barrier solver, and nonlinear solvers, respectively. The barrier solver, also known as the interior point solver, and the nonlinear solver are optional features and require additional purchases.

Output Arguments:

Name	Description
pnStatus	A pointer to an integer variable containing the status of the optimization. For possible values, refer to the *Common Macro Definitions* table.

Remarks:

- The solver returns *LS_STATUS_INFORUNB* as solution status if primal or dual model is found to be infeasible. This could be determined either by the presolver or by phase-1 of dual simplex (*LS_METHOD_DSIMPLEX*) provided the dual is infeasible. In the former case, the solver computes no solutions and hence all solution access routines, such as LSgetPrimalSolution, returns an *LSERR_NOT_AVAILABLE* error. However, the solver can be forced to compute a basis by setting *LS_IPARAM_SOLVER_IUSOL* to 1. In the second case, the actual status of the primal model can be found by re-optimizing the model using the primal simplex method (*LS_METHOD_PSIMPLEX*).
- LINDO API is equipped with advanced recovery techniques that resolve numeric issues stemming from

  (a) Poor scaling,
  (b) Linear dependency among model variables (columns).
  (c) Degeneracy (redundancies in the formulation) in primal and or dual space.

  In rare pathological instances, it is possible that the solver returns a *LSERR_NUMERIC_INSTABILITY* error using the default tolerance setting. In this case, accumulated errors that occurred during numeric computations were so severe that the solver could not take further steps towards optimality. For all such cases, however, there exist a certain tolerance settings that would render the model solvable. The main tolerances that affect the numerical properties are primal and dual feasibility tolerances. The latter is also known as the optimality tolerance.
- If the LS_METHOD_BARRIER is used, a crossover to a basic solution is done at the end. If, instead, you want the nonbasic interior point solution, then use *LSsetModIntParameter*() to set the parameter LS_IPARAM_SOLVER_IPMSOL=1.
- Prior to solving the problem, LS_IPARAM_DECOMPOSITION_TYPE parameter can be set to LS_LINK_BLOCKS_NONE to force the linear solver to exploit total decomposition.
- The solution process can be lengthy on large models. LINDO API can be set to periodically callback to your code to allow you to monitor the solver's progress. For more information, see *LSsetCallback*().
- To solve mixed-integer models, see *LSsolveMIP*().

LSsolveGOP()

Description:

Optimizes a global optimization problem.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*.

Prototype:

int	LSsolveGOP(pLSmodel pModel, int *pnStatus)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
pnStatus	A pointer to an integer variable containing the status of the optimization. For possible values, refer to the *Common Macro Definitions* table.

Remarks:

- LINDO API's global optimization solver is specifically designed to solve hard nonlinear models with multiple local solutions.
- See the *Global Optimization Parameters* section above for available parameters that could be used to fine tune the global optimizer to yield improved performance in solving different problem classes.
- The solution process can be lengthy on medium to large models with multiple extrema. LINDO API can be set to periodically callback to your code to allow you to monitor the solver's progress. For more information, see *LSsetCallback*() and *LSsetMIPCallback*().

LSsolveMIP()

Description:

Optimizes a mixed-integer programming model using branch-and-cut.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSsolveMIP(pLSmodel pModel, int *pnStatus)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
pnStatus	A pointer to an integer variable containing the status of the optimization. For possible values, refer to the *Common Macro Definitions* table.

Remarks:

- To solve continuous models, see *LSoptimize*().
- To establish branching priority amongst the variable, see *LSloadVarPriorities*().
- The solution process can be lengthy on large models. LINDO API can be set to periodically callback to your code to allow you to monitor the solver's progress. For more information, see *LSsetCallback*() and *LSsetMIPCallback*().
- Prior to solving the problem, LS_IPARAM_DECOMPOSITION_TYPE parameter can be set to LS_LINK_BLOCKS_NONE to force the mixed-integer solver to exploit total decomposition.
- *LSbnbSolve*(), from LINDO API 1.x, has been deprecated. LINDO API is equipped with a state-of-the-art MIP (LP) presolver that performs a wide range of reduction and elimination techniques that aims at reducing the size of a given problem before optimizing it. The preprocessing phase terminates with one of the following outputs,

  1) A reduced model ready to be submitted to the solver engine.
  2) A solution status indicating infeasibility (LS_STATUS_INFEASIBLE)
  3) A solution status indicating unboundedness (LS_STATUS_UNBOUNDED)
  4) A solution status indicating infeasibility or unboundedness (LS_STATUS_INFORUNB), but no certificate of which.

LSsolveSP ()

Description:

Solves the SP models. All parameters controlling the solver should be set before calling the routine.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsolveSP (pLSmodel pModel, int * pnStatus)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
pnStatus	An integer reference for the status

Solution Query Routines

The routines in this section allow you to retrieve information regarding a model's solution values following optimization.

Note: LINDO API requires that sufficient memory is allocated for each output argument of the retrieving function.

LSgetBasis()

Description:

Gets information about the basis that was found after optimizing the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetBasis(pLSmodel pModel, int *panCstatus, int *panRstatus)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
panCstatus	A pointer to an integer vector in which information about the status of the variables is to be placed. The length of this vector must be ≥ the number of variables in the model. The *i*-th element of this array returns information on the *i*-th variable as follows: ≥0: Index of the row that variable is basic in −1: Nonbasic at lower bound −2: Nonbasic at upper bound −3: Free and nonbasic at zero value This value may be set to NULL if column basis information is not needed.
panRstatus	A pointer to an integer vector in which information about the status of the constraints is to be placed. The length of this vector must be ≥ the number of constraints in the model. The *i*-th element of this array returns information on the *i*-th constraint as follows: ≥0: Row's associated slack variable is basic −1: Row's associated slack variable is nonbasic at zero This value may be set to NULL if constraint information is not needed.

Remarks

- To load a basis, use *LSloadBasis*().
- *LSloadBasis*() does not require the row indices that the variables are basic in. Setting all basic variables to a nonnegative integer is sufficient to specify a basis.
- *LSgetBasis*(), in addition to the indices of basic variables, returns also the indices of the rows that variables are basic in.
- If the LP presolver was on during LP optimization, the column status of basic variables that were eliminated from the original LP will not correspond to row indices. In order to obtain the row indices of all the basic variables, you will need to turn off the LP presolver and call *LSoptimize()* again. This reoptimization would normally take zero iteration because the last basis is already optimal. Calling *LSgetBasis()* after the reoptimization would return panCstatus with correct row indices for all basic columns.

Note: Solution query routines will return an error code of 2009 -the requested info not available- whenever they are called after the optimization halts without a solution being computed. The main reasons for not having a solution after optimization are

1) optimization halts due to a time or iteration limit

2) optimization halts due to numerical errors

3) optimization halts due to CTRL-C (user break)

4) presolver determines the problem to be infeasible or unbounded

5) the solver used in current optimization session (e.g. LSsolveMIP) did not produce any results for the queried solution object (e.g. GOP solution).

The last error code returned by the optimizer can be retrieved by calling LSgetInfo() function.

LSgetDualSolution()

Description:

Returns the value of the dual variables for a given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetDualSolution(pLSmodel pModel, double *padDual)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padDual	A pointer to a double precision vector in which the dual solution is to be placed. The length of this vector must be equal to or exceed the number of constraints in the model. It is

	assumed that sufficient memory has been allocated for this vector.

Remarks:

- The dual variable associated with a constraint is the increase in the objective function value per unit of increase in the right-hand side of the constraint, given the change is within the sensitivity limits of that RHS. Thus, if the objective is MAX, then a "≤" constraint has a nonnegative dual price and a "≥" constraint has a nonpositive dual price. If the objective is MIN, then a "≤" constraint has a nonpositive dual price and a "≥" constraint has a nonnegative dual price.
- To learn more about sensitivity analysis, see Chapter 10.
- To get slack values on the constraints, see *LSgetSlacks*().

LSgetInfo()

Description:

Returns model or solution information about the current state of the LINDO API solver after model optimization is completed. This function cannot be used to access callback information.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetInfo(pLSmodel pModel, int nQuery, void *pvValue)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *Lsmodel.*
nQuery	For possible values, refer to the tables under '*Available Information*' section.

Output Arguments:

Name	Description
pvValue	This is a pointer to a memory location where LINDO API will return the requested information. You must allocate sufficient memory for the requested information prior to calling this function.

Remarks:

- This function cannot be used to access callback information. *LSgetCallbackInfo()* should be used instead.
- Query values whose names begin with LS_IINFO return integer values, while those whose names begin with LS_DINFO return double precision floating point values.

LSgetMIPBasis()

Description:

Gets information about the basis that was found at the node that yielded the optimal MIP solution.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetMIPBasis(pLSmodel pModel, int *panCstatus, int *panRstatus)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
panCstatus	A pointer to an integer vector in which information about the status of the variables is to be placed. The length of this vector must be ≥ the number of variables in the model. The *i*-th element of this array returns information on the *i*-th variable as follows: ≥0: Index of row that variable is basic in -1: Nonbasic at lower bound -2: Nonbasic at upper bound -3: Free and nonbasic at zero value This value may be set to NULL if column basis information is not needed..
panRstatus	A pointer to an integer vector in which information about the status of the constraints is to be placed. The length of this vector must be ≥ the number of constraints in the model. The *i*-th element of this array returns information on the *i*-th constraint as follows: 0: Slack is basic -1: Slack is nonbasic at zero This value may be set to NULL if constraint information is not needed.

Remarks:

- For information on loading a mixed-integer program's formulation data into the system, see *LSloadVarType*().

LSgetMIPDualSolution()

Description:

Gets the current dual solution for a MIP model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetMIPDualSolution(pLSmodel pModel, double *padDual)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padDual	A pointer to a double precision vector containing the optimal dual solution to the LP relaxation of a MIP model by fixing all integer variables with respect to the resident MIP solution. The number of elements in this vector must equal, or exceed, the number of constraints in the model.

Remarks:

- For information on loading a mixed-integer program's formulation data into the system, see *LSloadVarType*().

LSgetMIPPrimalSolution()

Description:

Gets the current primal solution for a MIP model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetMIPPrimalSolution(pLSmodel pModel, double *padPrimal)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padPrimal	A pointer to a double precision vector in which the primal solution to the integer model is to be placed. The length of this

	vector is equal to or exceeds the number of variables in the model—continuous and integer.

Remarks:

- For information on loading a mixed-integer program's formulation data into the system, see *LSloadVarType*().
- To get the solution for a continuous model, see *LSgetPrimalSolution*().

LSgetMIPReducedCosts()

Description:

Gets the current reduced cost for a MIP model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetMIPReducedCosts(pLSmodel pModel, double *padRedCostl)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padRedCostl	A pointer to a double precision vector containing the optimal reduced costs to the LP relaxation of a MIP model by fixing all integer variables with respect to the resident MIP solution. The number of elements in this vector must equal, or exceed, the number of constraints in the model.

Remarks:

- For information on loading a mixed-integer program's formulation data into the system, see *LSloadVarType*().

LSgetMIPSlacks()

Description:

Gets the slack values for a mixed-integer model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetMIPSlacks(pLSmodel pModel, double *padSlacks)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padSlacks	A pointer to a double precision vector in which the slack values are to be placed. The number of elements in this vector must equal, or exceed, the number of constraints in the model.

Remarks:

- The ability to solve mixed-integer programs is an optional feature. Not all installations will have this capability. To determine if your license includes MIP functionality, use *LSgetModelIntParameter*() with license information access macros.
- To get the slacks on a continuous LP model, see *LSgetSlacks*().

LSgetPrimalSolution()

Description:

Returns the primal solution values for a given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetPrimalSolution(pLSmodel pModel, double *padPrimal)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padPrimal	A pointer to a vector in which the primal solution is to be placed. The length of this vector must equal or exceed the number of variables in the model.

Remarks:

- To get reduced costs on the variables, see *LSgetReducedCosts*().

LSgetReducedCosts()

Description:

Returns the reduced cost of all variables of a given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetReducedCosts(pLSmodel pModel, double *padRedcosts)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padRedcosts	A pointer to a double precision vector in which the reduced costs of the variables are to be returned. The vector length must be equal to or exceed the number of variables in the model.

Remarks:

- The reduced cost is the dual price of the simple lower or upper bound constraint of a variable. Thus, if the objective is MIN, then a binding lower bound will have a positive reduced cost, and a binding upper bound will have a negative reduced cost. If the objective is MAX, then a binding lower bound will have a negative reduced cost, and a binding upper bound will have a positive reduced cost.
- To get primal values on the variables, see *LSgetPrimalSolution*().

LSgetReducedCostsCone()

Description:

Returns the reduced cost of all cone variables of a given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetReducedCostsCone(pLSmodel pModel, double *padRedcosts)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padRedcosts	A pointer to a double precision vector in which the reduced costs of the variables are to be returned. The vector length must be equal to or exceed the number of variables in the model.

LSgetSlacks()

Description:

Returns the value of the slack variable for each constraint of a continuous model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetSlacks(pLSmodel pModel, double *padSlacks)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padSlacks	A pointer to a double precision vector in which the slack variables are to be returned. The length of this vector must equal or exceed the number of constraints in the model. Slack values are computed using the formula: $s = b – Ax$, where s is the vector of slacks, b is the right-hand side vector, A is the nonzero matrix for the basic columns, and x is the solution vector. Thus, less-than-or-equal-to constraints will return nonnegative values when feasible, while greater-than-or-equal-to constraints will return nonpositive values when feasible.

Remarks:

- To get dual values of the constraints, see *LSgetDualSolution*().
- To get the slacks for a MIP model, see *LSgetMIPSlacks*().

LSgetSolution()

Description:

Gets the solution specified by the second argument,

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetSolution(pLSmodel pModel, int nWhich, double *padValues)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nWhich	An integer parameter specifying the solution to be retrieved. Possible values are: • LSSOL_BASIC_PRIMAL • LSSOL_BASIC_DUAL • LSSOL_BASIC_SLACK • LSSOL_BASIC_REDCOST • LSSOL_INTERIOR_PRIMAL • LSSOL_INTERIOR_DUAL • LSSOL_INTERIOR_SLACK • LSSOL_INTERIOR_REDCOST

Output Arguments:

Name	Description
padValues	A pointer to a double precision vector in which the specified solution is to be placed. The length of this vector must be equal to or exceed the number of elements to be retrieved (e.g. number of constraints or variables). It is assumed that sufficient memory has been allocated for this vector.

LSgetNodePrimalSolution ()

Description:

Returns the primal solution for the specified node.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetNodePrimalSolution (pLSmodel pModel, int iScenario, int iStage, double * padX)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario the node belongs to.
iStage	An integer specifying the stage the node belongs to.
padX	A double array to return specified nodes's dual solution The length of this vector is equal to the number of variables in the stage associated with the node. It is assumed that memory has been allocated for this vector.

Remarks:

The number of variables or constraints in a stage can be accessed via **LSgetStocInfo()**.

LSgetScenarioObjective ()

Description:

Returns the objective value for the specified scenario.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetScenarioObjective (pLSmodel pModel, int iScenario, double * pObj)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario index.
pObj	A reference to a double variable to return the result.

LSgetScenarioPrimalSolution ()

Description:

Returns the primal solution for the specified scenario.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetScenarioPrimalSolution (pLSmodel pModel, int iScenario, double * padX, double * pObj)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario index.
padX	A double array to return scenario's primal solution. The length of this vector is equal to the number of variables in the core model. It is assumed that memory has been allocated for this vector.
pObj	A reference to a double to return the objective value for the specified scenario.

LSgetScenarioReducedCost ()

Description:

Returns the reduced cost for the specified scenario.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetScenarioReducedCost (pLSmodel pModel, int iScenario, double * padD)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario index.
padD	A double array to return scenario's reduced cost. The length of this vector is equal to the number of variables in the core model. It is assumed that memory has been allocated for this vector.

LSgetNodeDualSolution ()

Description:

Returns the dual solution for the specified node.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetNodeDualSolution (pLSmodel pModel, int iScenario, int iStage, double * padY)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario the node belongs to
iStage	An integer specifying the stage the node belongs to.
padY	A double array to return specified nodes's dual solution The length of this vector is equal to the number of constraints in the stage associated with the node. It is assumed that memory has been allocated for this vector.

LSgetNodeSlacks ()

Description:

Returns the dual solution for the specified node.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetNodeSlacks (pLSmodel pModel, int iScenario, int iStage, double * padS)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario the node belongs to
iStage	An integer specifying the stage the node belongs to.
padS	a double array to return specified nodes's dual solution The length of this vector is equal to the number of constraints in the stage associated with the node. It is assumed that memory has been allocated for this vector.

LSgetScenarioDualSolution ()

Description:

Returns the dual solution for the specified scenario.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetScenarioDualSolution (pLSmodel pModel, int iScenario, double * padY)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario index.
iStage	An integer specifying the stage the node belongs to.
padY	A double array to return scenario's dual solution The length of this vector is equal to the number of constraints in the core model. It is assumed that memory has been allocated for this vector.

LSgetScenarioSlacks ()

Description:

Returns the primal slacks for the specified scenario.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetScenarioSlacks (pLSmodel *pModel*, int *iScenario*, double * *padS*)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iScenario	An integer specifying the scenario index.
iStage	An integer specifying the stage the node belongs to.
padS	A double array to return scenario's primal slacks. The length of this vector is equal to the number of constraints in the core model. It is assumed that memory has been allocated for this vector.

LSgetNextBestMIPSoln()

Description:

Generates the next best (in terms of objective value) solution for the current mixed-integer model. Repeated calls to *LSgetNextBestMIPSoln()* will allow one to generate the so-called *K-Best* solutions to mixed-integer model. This is useful for revealing alternate optima.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetNextBestMIPSoln(pLSmodel pModel, int *pnIntModStatus)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
pnIntModStatus	A pointer to an integer varaible that will return the status on the new, next-best solution.

Remarks:

- *LSgetNextBestMIPSoln()* may not be used on models containing general integer variables; all integer variables must be binary.
- *LSgetNextBestMIPSoln()* modifies the original, base model by adding one constraint to the end of the model for each call to *LSgetNextBestMIPSoln()*. To return to the original model, you must delete these additional constraints after the final call to *LSgetNextBestMIPSoln()*.
- To generate the K-Best solutions for a MIP, one would perform the following steps:

    1. Generate the base MIP model.
    2. Call *LSsolveMIP()* to optimize the base model.
    3. Set i=0.
    4. If current solution status is not optimal, go to step 10.
    5. Call one or more model solution query routines to retrieve the current solution.
    6. Set i=i+1.
    7. If i>=K go to 10.
    8. Call *LSgetNextBestMIPSoln()* to find the next best solution.
    9. Go to step 4.
    10. Exit.

Model Query Routines

The routines in this section allow you to retrieve the components of the model data.

LSgetConeDatai()

Description:

Retrieve data for cone *i*.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetConeDatai(pLSmodel pModel, int iCone, char *pachConeType, int *piNnz, int *piCols)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
iCone	The index of the cone to retrieve the data for.

Output Arguments:

Name	Description
pachConeType	A pointer to a character variable that returns the constraint's type. The returned value will be ''Q', or 'R'.
piNnz	A pointer to an integer variable that returns the number of variables characterizing the cone.
piCols	A pointer to an integer vector that returns the indices of variables characterizing the cone.

LSgetConeIndex()

Description:

Gets the index of a cone with a specified name.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetConeIndex(pLSmodel pModel, char *pszConeName, int *piCone)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
pszConeName	A pointer to a null-terminated string containing the name of the cone for which the index is requested.

Output Arguments:

Name	Description
piCone	A pointer to an integer scalar that returns the index of the cone requested.

LSgetConeNamei()

Description:

Gets the name of a cone with a specified index.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetConeNamei(pLSmodel pModel, int iCone, char *pachConeName)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
iCone	Index of the cone whose name is to be retrieved.

Output Arguments:

Name	Description
pachConeName	A pointer to a character array that contains the cone's name with a null terminator.

LSgetConstraintDatai()

Description:

Gets data on a specified constraint.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetConstraintDatai(pLSmodel pModel, int iCon, char *pchContype, char *pchIsNlp, double *pdB)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
iCon	The index of the constraint you wish to receive information on.

Output Arguments:

Name	Description
pchContype	A pointer to a character variable that returns the constraint's type. The returned value will be 'L', 'E', 'G', or 'N', for less-than-or-equal-to, equal to, greater-than-or-equal-to, or neutral, respectively.
pchIsNlp	A pointer to a character that returns 0 if the constraint is linear and 1 if it is nonlinear.
pdB	A pointer to a double precision variable that returns the constraint's right-hand side value.

LSgetConstraintIndex()

Description:

Gets the index of a constraint with a specified name.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetConstraintIndex(pLSmodel pModel, char *pszConname, int *piCon)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
pszConname	A pointer to a character array that contains the constraint's name with a null terminator.

Output Arguments:

Name	Description
piCon	A pointer to an index of the constraint whose name is to be retrieved.

LSgetConstraintNamei()

Description:

Gets the name of a constraint with a specified index.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetConstraintNamei(pLSmodel pModel, int iCon, char *pszConname)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
iCon	Index of the constraint whose name is to be retrieved.

Output Arguments:

Name	Description
pszConname	A pointer to a character array that contains the constraint's name with a null terminator.

LSgetLPConstraintDatai()

Description:

Retrieves the formulation data for a specified constraint in a linear or mixed integer linear program. Individual pointers may be set to NULL if a particular item is not required.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetLPConstraintDatai(pLSmodel pModel, int iCon, char *pchContype, double *pdB, int *pnNnz, int *paiVar, double *padAcoef)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
iCon	An integer containing the index of the constraint whose data you wish to retrieve.

Output Arguments:

Name	Description
pchContype	A pointer to a character that returns the constraint's type. Values returned are 'L' for less-than-or-equal-to, 'E' for equal-to, 'G' for greater-than-or-equal-to, or 'N' for neutral.
pdB	A pointer to a double precision quantity that returns the constraint's right-hand side coefficient.
pnNnz	A pointer to an integer that returns the number of nonzero coefficients in the constraint.
paiVar	A pointer to an integer array that returns the indices of the variables with nonzero coefficients in the constraint. You must allocate all required space for this array before calling this routine.
padAcoef	A pointer to a double precision array that returns the constraint's nonzero coefficients. You must allocate all required space for this array before calling this routine.

Remarks:

- If you know a constraint's name, but don't know its internal index, you can obtain the index with a call to *LSgetConstraintIndex*(). To get a constraint's name, given its index, see *LSgetConstraintNamei*().

LSgetLPData()

Description:

Retrieves the formulation data for a given linear or mixed integer linear programming model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetLPData(pLSmodel pModel, int *pObjsense, double *pdObjconst, double *padC, double *padB, char *pachContypes, int *paiAcols, int *pacAcols, double *padAcoef, int *paiArows, double *padL, double *padU)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
pObjsense	A pointer to an integer indicating whether the objective is to be maximized or minimized. Valid values are LS_MAX or LS_MIN, respectively.
pdObjconst	A pointer to a double precision constant to be added to the objective value.
padC	A pointer to a double precision vector that returns the linear program's objective coefficients. This vector must have at least one element for each variable in the model.
padB	A pointer to a double precision vector that returns the constraint right-hand side coefficients. This vector must have at least one element for each constraint in the model.
pachContypes	A pointer to a character vector that returns the type of each constraint. Values returned are 'L', 'E', 'G', or 'N' for less-than-or-equal-to, equal-to, greater-than-or-equal-to, or neutral, respectively. This array must contain at least one byte for each constraint.
paiAcols	A pointer to an integer vector returning the index of the first nonzero in each column. This vector must have $n + 1$ entries, where n is the number of variables in the model. The last entry will be the index of the next appended column, assuming one was to be appended.
pacAcols	A pointer to an integer vector returning the length of each column. Note that the length of a column can be set to be greater than the values of paiAcols would suggest. In other words, it is possible for pacAcols[i] < paiAcols[i+1] –

	paiAcols[*i*].
padAcoef	A pointer to a double precision vector returning the nonzero coefficients of the constraint matrix. This vector must contain at least one element for each nonzero in the constraint matrix.
paiArows	A pointer to an integer vector returning the row indices of the nonzeros in the constraint matrix. You must allocate at least one element in this vector for each nonzero in the constraint matrix.
padL	A pointer to a double precision vector containing the lower bound of each variable. If there is no lower bound on the variable, then this value will be equal to -LS_INFINITY. You must allocate at least one element in this vector for each variable in the model.
padU	A pointer to a double precision vector containing the upper bound of each variable. If there is no upper bound on the variable, then this value will be equal to LS_INFINITY. You must allocate at least one element in this vector for each variable in the model.

Remarks:

- For information on loading a linear program's formulation data into the system, see *LSloadLPData*().
- Pointers may be set to NULL for any information not required.

LSgetLPVariableDataj()

Description:

Retrieves the formulation data for a specified variable. Individual pointers may be set to NULL if a particular item is not required.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetLPVariableDataj(pLSmodel pModel, int iVar, char *pchVartype, double *pdC, double *pdL, double *pdU, int *pnAnnz, int *paiArows, double *padAcoef)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
iVar	An integer containing the index of the variable whose data you wish to retrieve.

Output Arguments:

Name	Description
pchVartype	A pointer to a character that returns the variable's type. Values returned are 'B' for binary, 'C' for continuous, or 'I' for general integer.
pdC	A pointer to a double precision quantity that returns the variable's objective coefficient.
pdL	A pointer to a double precision quantity that returns the variable's lower bound.
pdU	A pointer to a double precision quantity that returns the variable's upper bound.
pnAnnz	A pointer to an integer that returns the number of nonzero constraint coefficients in the variable's column.
paiArows	A pointer to an integer array that returns the row indices of the variable's *pnAnnz nonzeros. You must allocate the required space for this array before calling this routine.
padAcoef	A pointer to a double precision array that returns the variable's nonzero coefficients. You must allocate all required space for this array before calling this routine.

Remarks:

- If you know a variable's name, but don't know its internal index, you can obtain the index with a call to *LSgetVariableIndex*(). To get a variable's name given its index, see *LSgetVariableNamej*().

LSgetNameData()

Description:

Returns the names—objective, right-hand side vector, range vector, bound vector, constraints, and variables—of a given model. Any of the pointers to the names can be input as NULL if the corresponding information is not required.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetNameData(pLSmodel pModel, char *pszTitle, char *pszObjname, char *pszRhsname, char *pszRngname, char *pszBndname, char **paszConnames, char * pachConNameData , char **paszVarnames, char *pachVarNameData)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
pszTitle	A pointer to a character array that returns the title of the problem. A model's title can be of any length, so be sure to allocate sufficient space to store the title you originally passed to LINDO API. The returned title will be null terminated.
pszObjname	A pointer to a character array that will return the name of the objective, null terminated.
pszRhsname	A pointer to a character array that returns the name of the right-hand side vector, null terminated.
pszRngname	A pointer to a character array that returns the name of the range vector, null terminated. This pointer is reserved for future use.
pszBndname	A pointer to a character array that returns the name of the bound vector, null terminated.
paszConnames	A pointer to an array of pointers of length equal to or exceeding the number of constraints. On return, these pointers will point to the constraint names stored in the character array pointed to by *paszConNameData*. You must allocate space for *m* pointers, where *m* is the number of rows.
pachConNameData	A pointer to an array of characters used to store the actual constraint name data.
paszVarnames	A pointer to an array of pointers of length equal to or exceeding the number of variables. On return, the pointers will

	point to the variable names stored in the character array pointed to by *paszVarNameData*. You must allocate space for *n* pointers, where *n* is the number of variables.
pachVarNameData	A pointer to an array of characters used to store the actual variable name data.

Remarks:

- The right-hand side name, range name, and bound name are typically only used if the model was read from an MPS file.
- You may set any of the pointers to NULL if the particular name data is not relevant.
- The constraint and variable name data in the output arguments *pachConNameData* and *pachVarNameData* are created internally by LINDO API with *LSloadNameData*.

LSgetNLPConstraintDatai()

Description:

Gets data about the nonlinear structure of a specific row of the model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetNLPConstraintDatai (pLSmodel pModel, int i, int *pnNnzi, int *paiColi, double *padCoefi);

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
i	An integer indicating the constraint to retrieve the data for.

Output Arguments:

Name	Description
pnNnzi	A pointer to an integer returning the number of nonlinear nonzeros in constraint *i*.
paiColi	A pointer to an integer vector returning the column indices of the nonlinear nonzeros in the i^{th} row of the constraint matrix.
padCoefi	A pointer to a double precision vector returning the current values of the nonzero coefficients in the i^{th} row of the coefficient (Jacobian) matrix.

Remarks:

- It is the caller's responsibility to make sure that the vectors *paiColi* and *padCoefi* have room for at least **pnNnzi* elements.

LSgetNLPData()

Description:

Gets data about the nonlinear structure of a model, essentially the reverse of *LSloadNLPData*().

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetNLPData(pLSmodel pModel, int *paiCols, int *pacCols, double *padCoef, int *paiRows, int *pnObj, int *paiObj, double *padObjCoef, char *pachConType)

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
paiCols	A pointer to an integer vector returning the index of the first nonlinear nonzero in each column. This vector must have *nVars*+1 entries, where *nVars* is the number of variables. The last entry will be the index of the next appended column, assuming one was to be appended.
pacCols	A pointer to an integer vector returning the number of nonlinear elements in each column.
padCoef	A pointer to a double precision vector returning the current values of the nonzero coefficients in the (Jacobian) matrix. This can be NULL.
paiRows	A pointer to an integer vector returning the row indices of the nonlinear nonzeros in the coefficient matrix.
pnObj	An integer returning the number of nonlinear variables in the objective function.
paiObj	A pointer to an integer vector returning column indices of the nonlinear terms in the objective.
padObjCoef	A pointer to a double precision vector returning the current partial derivatives of the objective corresponding to the variables *paiObj* [ ].
pachConType	A pointer to a character vector whose elements indicate whether a constraint has nonlinear terms or not. If *pachConType* [*i*] > 0, then constraint *i* has nonlinear terms.

LSgetNLPObjectiveData()

Description:

Gets data about the nonlinear structure of the objective row.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetNLPObjectiveData (pLSmodel pModel, int *pnObj, int *paiObj, double *padObjCoef);

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
pnObj	A pointer to an integer returning the number of nonlinear variables in the objective function.
paiObj	A pointer to an integer vector returning column indices of the nonlinear terms in the objective.
padObjCoef	A pointer to a double precision vector returning the current partial derivatives of the objective corresponding to the variables in *paiObj* with respect to the last primal solution computed during the iterations

Remarks:

- It is the caller's responsibility to make sure that the vectors *paiObj* and *padObjCoef* have room for at least **pnObj* elements.

LSgetNLPVariableDataj()

Description:

Gets data about the nonlinear structure of a specific variable of the model

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetNLPVariableDataj (pLSmodel pModel,int j, int *pnNnzj, int *paiRowj, double * padCoefj);

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
j	An integer indicating the column to retrieve the data for.

Output Arguments

Name	Description
pnNnzj	A pointer to an integer returning the number of nonlinear nonzeros in column *j*.
paiRowj	A pointer to an integer vector returning the row indices of the nonlinear nonzeros in the j^{th} column of the constraint matrix.
padCoefj	A pointer to a double precision vector returning the current values of the nonzero coefficients in the j^{th} column of the coefficient (Jacobian) matrix with respect to the last primal solution computed during the iterations.

Remarks:

- It is the caller's responsibility to make sure that the vectors *paiRowj* and *padCoefj* have room for at least **pnNnzj* elements.

LSgetQCData()

Description:

Retrieves the quadratic data from an *LSmodel* data structure. Any of the pointers in the output argument list can be set to NULL if the corresponding information is not required.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error codes.*

Prototype:

int	LSgetQCData(pLSmodel pModel, int *paiQCrows, int *paiQCcols1, int *paiQCcols2, double *padQCcoef)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* from which to retrieve the problem data.

Output Arguments:

Name	Description
paiQCrows	A pointer to an integer vector containing the index of the constraint associated with each quadratic term with a nonzero coefficient. The objective row is indicated with an index of -1.This vector must have room for all nonzero entries.
PaiQCcols1	A pointer to an integer vector containing the index of the first variable defining each quadratic term. This vector must have one element for each nonzero in the matrix.
paiQCcols2	A pointer to an integer vector containing the index of the second variable defining each quadratic term. This vector must have one element for each nonzero in the matrix.
padQCcoef	A pointer to a double vector containing the nonzero coefficients in the quadratic matrix. This vector must also have space for each nonzero matrix element.

Remarks:

- *LSgetQCData* does not return the number of nonzeros in the Q matrices. You can get that information using *LSgetInfo*().

LSgetQCDatai()

Description:

Retrieves the quadratic data associated with constraint *i* from an *LSmodel* data structure. Any of the pointers in the output argument list can be set to NULL if the corresponding information is not required.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error codes*.

Prototype:

int	LSgetQCDatai(pLSmodel pModel, int iCon, int *pnQCnnz, int *paiQCcols1, int *paiQCcols2, double *padQCcoef)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* from which to retrieve the problem data.
iCon	An integer scalar specifying the constraint for which the quadratic data will be retrieved.

Output Arguments:

Name	Description
pnQCnnz	A pointer to an integer containing the number of nonzeros in the coefficient matrix of the quadratic term.
paiQCcols1	A pointer to an integer vector containing the index of the first variable defining each quadratic term. This vector must have one element for each nonzero in the matrix.
paiQCcols2	A pointer to an integer vector containing the index of the second variable defining each quadratic term. This vector must have one element for each nonzero in the matrix.
padQCcoef	A pointer to a double vector containing the nonzero coefficients in the quadratic matrix. This vector must also have space for each nonzero matrix element.

LSgetSemiContData()

Description:

Retrieves the semi-continuous data from an *LSmodel* data structure. Any of the pointers in the output argument list can be set to NULL if the corresponding information is not required.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error codes*..

Prototype:

int	LSgetSemiContData(pLSmodel pModel, int *piNvars, int *piVarndx, double *padl, double *padu)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* from which to retrieve the problem data.

Output Arguments:

Name	Description
piNvars	A pointer to an integer variable to return the number of semi-continuous variables.
piVarndx	A pointer to an integer vector to return the indices of semi-continuous variables.
padl	A pointer to a vector to return the lower bounds of semi-continuous variables.
padu	A pointer to a vector to return the upper bounds of semi-continuous variables.

LSgetSETSData()

Description:

Retrieves sets data from an *LSmodel* data structure. Any of the pointers in the output argument list can be set to NULL if the corresponding information is not required.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error codes*

Prototype:

int	LSgetSETSData(pLSmodel pModel, int *piNsets, int *piNtnz, char *pachSETtype, int *piCardnum, int *piNnz, int piBegset, int *piVarndx)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* from which to retrieve the problem data.

Output Arguments:

Name	Description
piNsets	A pointer to an integer variable to return the number of sets in the model.
piNtnz	A pointer to an integer variable to return the total number of variables in the sets.
pachSETtype	A pointer to a character array to return the type of sets in the model. The size of this array should be at least *(*piNsets)*
piCardnum	A pointer to an integer array to return the cardinalities of sets in the model. The size of this array should be at least *(*piNsets)*
piNnz	A pointer to an integer array to return the number of variables in each set in the model. The size of this array should be at least *(*piNsets)*
piBegset	A pointer to an integer array returning the index of the first variable in each set. This vector must have (*piNsets + 1) entries, where **piNsets* is the number of sets in the model. The last entry will be the index of the next appended set, assuming one was to be appended.
piVarndx	A pointer to an integer vector returning the indices of the variables in the sets. You must allocate at least one element in this vector for each <variable,set> tuple (i.e. at least *piNtnz elements are required.)

LSgetSETSDatai()

Description:

Retrieves the data for set *i* from an *LSmodel* data structure. Any of the pointers in the output argument list can be set to NULL if the corresponding information is not required.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error codes.*

Prototype:

int	LSgetSETSDatai(pLSmodel pModel, int iSet, char *pachSETType, int *piCardnum, int *piNnz, int *piVarndx)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel* from which to retrieve the problem data.
iSet	The index of the set to retrieve the data for.

Output Arguments:

Name	Description
pachSETType	A pointer to a character variable to return the set type.
piCardnum	A pointer to an integer variable to return the set cardinality.
piNnz	A pointer to an integer variable to return the number of variables in the set.
piVarndx	A pointer to an integer vector to return the indices of the variables in the set. This vector should have at least *(*piNnz)* elements.

LSgetVariableIndex()

Description:

Retrieves the internal index of a specified variable name.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetVariableIndex(pLSmodel pModel, char *pszVarname, int *piVar)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
pszVarname	A pointer to a null terminated character string containing the name of the variable.

Output Arguments:

Name	Description
piVar	A pointer to an integer that returns the variable's index.

Remarks:

- To get a variable's name given its index, see *LSgetVariableNamej*().
- If you have problems with this routine, watch out for embedded blanks. For example, `"X005 "` (four trailing blanks) is not the same as `" X005"` (four leading blanks), is not the same as `"X005"` (no blanks).
- Refer to *LSreadMPSFile*() for a detailed description of the internal formatting of the name data.

LSgetVariableNamej()

Description:

Retrieves the name of a variable, given its index number.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetVariableNamej(pLSmodel pModel, int iVar, char *pszVarname)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
iVar	An integer containing the index of the variable whose name you wish to retrieve.

Output Arguments:

Name	Description
pszVarname	A pointer to a character array that returns the variable's name with a null terminator.

Remarks:

- To get a variable's formulation data given its index, see *LSgetLPVariableDataj*().

LSgetVarStartPoint()

Description:

Retrieves the values of the initial primal solution.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetVarStartPoint(pLSmodel pModel, double *padPrimal)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padPrimal	A pointer to a double precision vector containing starting values for each variable in the given model. The length of this vector is equal to the number of variables in the model.

LSgetVarType()

Description:

Retrieves the variable types and their respective counts in a given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetVarType(pLSmodel pModel, char *pachVartypes)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
pachVartypes	A pointer to a vector returning the type of each variable. Return value for each variable is either 'C' for a continuous variable, 'B' for a binary variable, or 'I' for a general integer variable. The length of this vector must be at least that of the current number of variables in the model. This pointer can be set to NULL if the variable types are not required.

Remarks:

- For information on loading a mixed-integer program's formulation data into the system, see *LSloadVarType*().

LSgetStageName ()

Description:

Get stage name by index.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetStageName (pLSmodel pModel, int stageIndex, char * stageName)

Input Arguments:

Name	Description
pModel	A reference to an instance of LSmodel object.
stageIndex	An integer specifying the index of the stage to retrieve the name for.
stageName	A string to retrieve the stage name (max length is 255 characters).

LSgetStageIndex ()

Description:

Get index of stage by name.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetStageIndex (pLSmodel pModel, char * stageName, int * stageIndex)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
stageName	A string specifying the name of the stage to return the index for.
stageIndex	A reference to an integer to return the index of the stage.

LSgetStocParIndex ()

Description:

Get the index of stochastic parameter by name.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetStocParIndex (pLSmodel pModel, char * svName, int * svIndex)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
svName	A string specifying the name of the stochastic parameter to return the index for.
svIndex	A reference to an integer to return the index of the stochastic parameter.

LSgetStocParName ()

Description:

Get name of stochastic parameter by index.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetStocParName (pLSmodel pModel, int svIndex, char * svName)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
svIndex	A reference to an integer to return the index of the stochastic parameter.
svName	A string specifying the name of the stochastic parameter to return the index for.

LSgetScenarioName ()

Description:

Get scenario name by index.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetScenarioName (pLSmodel pModel, int jScenario, char * scenarioName)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
jScenario	An integer specifying the scenario index.
scenarioName	A string reference to return the name of the scenario (Max lengt 255 characters).

LSgetScenarioIndex ()

Description:

Get index of a scenario by its name.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetScenarioIndex (pLSmodel pModel, char * scenarioName, int * jScenario)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
scenarioName	A string specifying the name of the scenario to return the index for.
jScenario	A reference an integer to return the index of the scenario.

LSgetProbabilityByScenario ()

Description:

Returns the probability of a given scenario.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetProbabilityByScenario (pLSmodel pModel, int jScenario, double * dprob)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
jScenario	An integer specifying the scenario index.
dprob	A reference to a double to return the probabability of the scenario.

LSgetProbabilityByNode ()

Description:

Returns the probability of a given node in the stochastic tree.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetProbabilityByNode (pLSmodel pModel, int iNode, double * dprob)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iNode	An integer specifying the node index.
dprob	A reference to a double to return the probabability of the node.

LSgetDeteqModel ()

Description:

Get the deterministric equivalent (DEQ) of the SP model, building it if not existent.

Returns:

ideModel an instance of LSmodel object referring to the DEQ model

Prototype:

int	LSgetDeteqModel (pLSmodel pModel, int iDeqType, int * perrorcode)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
iDeqType	An integer specifying the DEQ type (implicit or explicit). Possible values are: LS_DETEQ_FREE LS_DETEQ_IMPLICIT LS_DETEQ_EXPLICIT
perrorcode	an reference to an integer to return the error code.

LSgetNodeListByScenario ()

Description:

Retrieves the indices of the nodes that belong to a given scenario.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetNodeListByScenario (pLSmodel pModel, int jScenario, int * pNodesOnPath, int * pnNodes)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
jScenario	An integer specifying the scenario index
pNodesOnPath	An integer array to return the node list constituting the scenario. The length of this vector is equal to the number of stages in the model. It is assumed that memory has been allocated for this vector.
pnNodes	An integer pointer to return the actual number of nodes on the scenario.

Remarks:

Also loads the nodes of the specified scenario into an internal buffer.

LSgetStocParOutcomes ()

Description:

Retrieve the outcomes of stochastic parameters for the specified scenario.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetStocParOutcomes (pLSmodel pModel, int jScenario, double * padVals, double * pdProbability)

Input Arguments:

Name	Description
pModel	A reference to an instance of *LSmodel* object.
jScenario	An integer specifying the scenario index. be at least the number of stochastic parameters in the model.
padVals	a double vector to return the values of stochastic parameters for the specified scenario. The length of this vector should be at least the number of stochastic parameters in the model.
pdProbability	probability of the scenario.

Remarks:

Total number of stochastic parameters could be retrived with LS_IINFO_NUM_SPARS.

LSgetStocParData ()

Description:

Retrieve the data of stochastic parameters.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetStocParData (pLSmodel *pModel*, int * *paiStages*, double * *padVals*)

Input Arguments:

Name	Description
pModel	A reference to an instance of LSmodel object.
paiStages	an integer vector to return the stages of stochastic parameters. The length of this vector should be at least the number of stochastic parameters in the model.
padVals	a double vector to return the values of stochastic parameters for the specified scenario. The length of this vector should be at least the number of stochastic parameters in the model.

Remarks:

Total number of stochastic parameters could be retrived with LS_IINFO_NUM_SPARS.

LSgetDiscreteBlocks ()

Description:

Gets the stochastic data for the discrete block event at specified index.

Returns:

errorcode An integer error code listed in Appendix A.

Prototype:

int	LSgetDiscreteBlocks (pLSmodel pModel, int iEvent, int * nDistType, int * iStage, int * nRealzBlock, double * padProbs, int * iModifyRule)

Input Arguments:

Name	Description
pModel	A reference to an instance of LSmodel object.
iEvent	An integer specifying the index of the discrete block event.
nDistType	A reference to an integer to return the distribution type of the event (optional).
iStage	A reference to an integer to return the stage index of the block event.
nRealzBlock	A reference to an integer to return the number of block realizations in the event.
padProbs	An double vector to return event probabilities. The length of this vector should be *nRealzBlock or more.
iModifyRule	A reference to an integer to return the flag indicating whether stochastic parameters update the core model by adding or replacing.

Remarks:

iEvent cannot be larger than the total number of discrete block events in the SP model. You can use LSgetStocInfo() or LSgetInfo() with LS_IINFO_STOC_NUM_EVENTS_BLOCK to retrieve the maximum possible value for iEvent .

LSgetDiscreteBlockOutcomes ()

Description:

Gets the outcomes for the specified block-event at specified block-realization index.

Returns:

errorcode An integer error code listed in Appendix A.

Prototype:

int	LSgetDiscreteBlockOutcomes (pLSmodel pModel, int iEvent, int iRealz, int * nRealz, int * paiArows, int * paiAcols, int * paiStvs, double * padVals)

Input Arguments:

Name	Description
pModel	A reference to an instance of LSmodel object.
iEvent	An integer specifying the index of the discrete block event.
iRealz	An integer specifying the index of a block realization in the specified block event.
nRealz	A reference to an integer to return the number of individual stochastic parameters consitituting the block realization iRealz.
paiArows	An integer vector to return the row indices of stochastic parameters. in the block realization iRealz . This vector should have *nRealz elements or more.
paiAcols	An integer vector to return the column indices of stochastic parameters. in the block realization iRealz . This vector should have *nRealz elements or more.
paiStvs	An integer vector to return the (instruction-list) indices of stochastic parameters. in the block realization iRealz . This vector should have *nRealz elements or more.
padVals	A double vector to return the values associated with the stochastic parameters listed in paiStvs or (paiArows,paiAcols) The length of this vector should be *nRealz or more.

Remarks:

Only one of the following, paiStvs or (paiArows,paiAcols) , will take sensible values on return. paiStvs should be used with instruction-based input, whereas (paiArows,paiAcols) should be used with matrix-based input. The argument(s) of the other group can be NULL.

iEvent cannot be larger than the total number of discrete block events in the SP model. You can use LSgetStocInfo() or LSgetInfo() to retrieve the maximum possible value for iEvent .

LSgetDiscreteIndep ()

Description:

Gets the stochastic data for the (independent) discrete stochastic parameter at the specified event index.

Returns:

errorcode An integer error code listed in Appendix A.

Prototype:

int	LSgetDiscreteIndep (pLSmodel pModel, int iEvent, int * nDistType, int * iStage, int * iRow, int * jCol, int * iStv, int * nRealizations, double * padProbs, double * padVals, int * iModifyRule)

Input Arguments:

Name	Description
pModel	A reference to an instance of LSmodel object.
iEvent	An integer specifying the index of the discrete independent event.
nDistType	A reference to an integer to return the distribution type of the event (optional).
iStage	A reference to an integer to return the stage index of the discrete-independent event.
iRow	A reference to an integer to return the row index of the stochastic parameter.
jCol	A reference to an integer to return the column index of the stochastic parameter.
iStv	A reference to an integer specifying the index of stochastic parameter in the instruction list.
nRealizations	A reference to an integer to return the number of all possible realizations for the stochastic parameter.
padProbs	A double vector to return the probabilities associated with the realizations of the stochastic parameter. The length of this vector should be *nRealizations or more.
padVals	A double vector to return the values associated with the realizations of the stochastic parameter. The length of this vector should be *nRealizations or more.
iModifyRule	A reference to an integer to return the flag indicating whether stochastic parameters update the core model by adding or replacing.

Remarks:

Only one of the following, iStvs or (iRow,jCol) , will take sensible values on return. iStvs should be used with instruction-based input, whereas (iRow,jCol) should be used with matrix-based input. The argument(s) of the other group can be NULL.

iEvent cannot be larger than the total number of discrete independent events in the SP model. You can use LSgetStocInfo() or LSgetInfo() with LS_IINFO_STOC_NUM_EVENTS_DISCRETE to retrieve the maximum possible value for iEvent .

LSgetSampleSizes ()

Description:

Retrieve the number of nodes to be sampled in all stages.

Returns:

errorcode An integer error code listed in Appendix A.

Prototype:

int	LSgetSampleSizes (pLSmodel pModel, int * panSampleSizes)

Input Arguments:

Name	Description
pModel	A reference to an instance of LSmodel object.
panSampleSizes	an integer vector to return the sample size per stage The length of this vector should be the number of stages in the model or more.

LSgetVariableStages ()

Description:

Retrieve the stage indices of variables.

Returns:

errorcode An integer error code listed in Appendix A.

Prototype:

int	LSgetVariableStages (pLSmodel pModel, int * panStage)

Input Arguments:

Name	Description
pModel	A reference to an instance of LSmodel object.
panStage	an integer vector to return the stage indices of variables in the core model. The length of this vector should be at least the number of variables in the core model.

LSgetHistogram ()

Description:

Retrieves the histogram for given data with given bin specs.

Returns:

errorcode An integer error code listed in Appendix A.

- LSERR_ILLEGAL_NULL_POINTER
- LSERR_NO_METHOD_LICENSE

Prototype:

int	LSgetHistogram(pLSmodel pModel, int nSampSize, double *padVals, double *padWeights, double dHistLow, double dHistHigh, int *pnBins, int *panBinCounts, double *padBinProbs, double *padBinLow, double *padBinHigh, double *padBinLeftEdge, double *padBinRightEdge)

Input Arguments:

Name	Description

pModel	An instance of LSmodel object.
nSampSize	An integer specifying the length of the input array.
padVals	A double reference to the input array
padWeights	A double reference to the input weights
dHistLow	A double scalar specifying the low end of the histogram
dHistHigh	A double scalar specifying the high end of the histogram
pnBins	An integer reference to specify (or return) the number of bins (optional).
panBinCounts	An integer array to return bin counts. Length of this array should at least be (*pnBins).
panBinProbs	A double array to return bin probabilities. Length of this array should at least be (*pnBins).
padBinLow	An double array to return bin lows. Length of this array should at least be (*pnBins).
padBinHigh	An double array to return bin highs. Length of this array should at least be (*pnBins).
padBinLeftEdge	An double array to return bins left edges. Length of this array should at least be (*pnBins).
padBinRightEdge	An double array to return bins right edges. Length of this array should at least be (*pnBins).

Remarks:

- Set dHistLow = dHistHigh on input for the module to choose a suitable pair for low and high values defining the histogram.
- Set pnBins to zero on input for the module to choose a suitable value for the number of bins. On return this value will contain the number of bins. If you set pnBins to zero on input, make sure to allocate at least nSampSize elements for panBinCounts, panBinProbs, padBinLow, padBinHigh arrays.

LSgetScenarioModel ()

Description:

Get a copy of the scenario model.

Returns:

scenModel An instance of pLSmodel containing the scenario model.

Prototype:

pLSmodel	LSgetScenarioModel(pLSmodel pModel, int jScenario, int *pnErrorcode)

Input Arguments:

Name	Description
pModel	A reference to an instance of LSmodel object.
jScenario	An integer specifying the scenario to retrieve.
pnErrorcode	A reference to an integer to return the error code.

Model Modification Routines

The routines in this section can modify the structure of a model on an incremental basis. For instance, these routines may be used to add and/or delete constraints and/or variables. After modifying a model, the LINDO API solver will restart using the remains of the last solution basis. Thus, after applying modest modifications to a model, re-solving should be relatively fast. These routines are intended for making minor modifications to a model. If you need to pass a new formulation, it is faster to use a routine such as *LSloadLPData*(), which is discussed above in the *Model Loading Routines* section.

LSaddCones()

Description:

Adds cones to a given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSaddCones(pLSmodel pModel, int nCone, char *pszConeTypes, char **pcConenames, int *paiConebegcol, int *paiConecols)

Input Arguments:

Name	Description
model	A pointer to an instance of *LSmodel*.
nCone	An integer containing the number of cones to append.
pszConeTypes	A pointer to a character array containing the type of each cone to be added to the model.
pcConenames	A pointer to a vector of pointers to null terminated strings containing the name of each new cone.
paiConebegcol	A pointer to an integer vector containing the index of the first variable in each new cone. This vector must have *nCone* +1 entries. The last entry should be equal to the number of variables in the added cones.
paiConecols	A pointer to an integer vector containing the indices of the variables in the new cones.

LSaddConstraints()

Description:

Adds constraints to a given model. If both constraints and variables need to be added to a model and adding the new information in row format is preferred, then this routine can be called after first calling *LSaddVariables*().

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSaddConstraints(pLSmodel pModel, int nNumaddcons, char *pachContypes, char **paszConnames, int *paiArows, double *padAcoef, int *paiAcols, double *padB)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nNumaddcons	An integer containing the number of constraints to append.
pachContypes	A pointer to a character array containing the type of each constraint to be added to the model. Valid values for each constraint are 'L', 'E', 'G', or 'N' for less-than-or-equal-to, equal-to, greater-than-or-equal-to, or neutral, respectively.
paszConnames	A pointer to a vector of pointers to null terminated strings containing the name of each new constraint.
paiArows	A pointer to an integer vector containing the index of the first nonzero element in each new constraint. This vector must have *nNumaddcons* +1 entries. The last entry should be equal to the number of nonzeros in the added constraints.
padAcoef	A pointer to a double precision vector containing the nonzero coefficients of the new constraints.
paiAcols	A pointer to an integer vector containing the column indices of the nonzeros in the new constraints.
padB	A pointer to a double precision vector containing the right-hand side coefficients for each new constraint.

Remarks:

- If, in addition, variables need to be added to a model, then *LSaddVariables*() must be called prior to this routine. The call to *LSaddVariables*() should pass NULL as the *paiAcols*, *padAcoef*, and *paiArows* arguments.
- If you need to load a new model, *LSloadLPData*() is a more efficient routine

LSaddSETS()

Description:

Adds sets to a given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSaddSETS(pLSmodel pModel, int nSETS, char *pszSETStypes, int *paiCARDnum, int *paiSETSbegcol, int *paiSETScols)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nSETS	An integer containing the number of sets to add.
pszSETStypes	A pointer to a character array containing the type of each set to be added to the model.
paiCARDnum	An integer array containing the cardinalities of the sets to be added.
paiSETSbegcol	A pointer to an integer vector containing the index of the first variable in each new set. This vector must have *nSETS* +1 entries. The last entry should be equal to the total number of variables in the new sets.
paiSETScols	A pointer to an integer vector containing the indices of the variables in the new sets.

LSaddVariables()

Description:

Adds variables to a given model. If both constraints and variables need to be added to a model and adding the new information in column format is preferred, then this routine can be called after first calling *LSaddConstraints*().

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSaddVariables(pLSmodel pModel, int nNumaddvars, char *pachVartypes, char **paszVarnames, int *paiAcols, int *pacAcols, double *padAcoef, int *paiArows, double *padC, double *padL, double *padU)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nNumaddvars	The number of variables to append to the model.
pachVartypes	A pointer to a character array containing the types of each variable to be added to the model. Valid values for each variable are 'B', 'C', or 'I' for binary, continuous, or general integer, respectively.
paszVarnames	A pointer to a vector of pointers to null terminated strings containing the name of each new variable.
paiAcols	A pointer to an integer vector containing the index of the first nonzero element in each new column. This vector must have *nNumaddvars*+1 entries. The last entry should be equal to the number of nonzeros in the new columns.
pacAcols	A pointer to a vector containing the length of each column. Note that the length of a column can be set to be shorter than the values of *paiAcols* would suggest (i.e., it is possible for *pacAcols*[*i*] < *paiAcols*[*i*+1] – *paiAcols*[*i*]). This may be desirable in order to prevent memory reallocations if rows are subsequently added to the model. If the length of each column *i* is equal to *paiAcols*[*i*+1] – *paiAcols*[*i*], then *pacAcols* can be set to NULL on input.
padAcoef	A pointer to a double precision vector containing the nonzero coefficients of the new columns.
paiArows	A pointer to an integer vector containing the row indices of the nonzeros in the new columns.
padC	A pointer to a double precision vector containing the objective coefficients for each new variable.
padL	A pointer to a double precision vector containing the lower bound of each new variable. If there is no lower bound on a variable, then the corresponding entry in the vector should be set to -LS_INFINITY. If *padL* is NULL, then the lower bounds are internally set to zero.
padU	A pointer to a double precision vector containing the upper bound of each new variable. If there is no upper bound on the variable, then this value should be set to LS_INFINITY. If *padU* is NULL, then the upper bounds are internally set to LS_INFINITY.

Remarks:

- If, in addition, constraints need to be added to a model and adding the new information in column format is preferred, then this routine can be called after first calling

LSaddConstraints(). The call to *LSaddConstraints*() should pass NULL as the *paiArows*, *padAcoef*, and *paiAcols* arguments.

- NULL may be passed for *paiAcols*, *padAcoef*, and *paiArows*.

LSaddQCterms()

Description:

Adds quadratic elements to the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSaddQCterms (pLSmodel pModel, int nQCnonzeros, int *paiQCconndx, int *paiQCvarndx1, *paiQCvarndx2, double *padQCcoef)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nQCnonzeros	The total number of nonzeros in quadratic coefficient matrices to be added.
paiQCconndx	A pointer to a vector containing the index of the constraint associated with each nonzero quadratic term. This vector must have *nQCnonzeros* entries.
paiQCvarndx1	A pointer to a vector containing the indices of the first variable defining each quadratic term. This vector must have *nQCnonzeros* entries.
paiQCvarndx2	A pointer to a vector containing the indices of the second variable defining each quadratic term. This vector must have *nQCnonzeros* entries.
padQCcoef	A pointer to a vector containing the nonzero coefficients in the quadratic matrix. This vector must also have *nQCnonzeros* entries.

LSaddNLPAj()

Description:

Adds NLP elements to the specified column for the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSaddNLPAj (pLSmodel pModel, int iVar1, int nRows, int *paiRows, double *padAj)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
iVar1	The index of the variable to which NLP elements will be added.
nRows	The total number of constraints for which NLP elements will be added.
paiRows	A pointer to an integer vector containing the row indices of the nonlinear elements. The indices are required to be in ascending order.
padAj	A pointer to a double vector containing the initial nonzero coefficients of the NLP elements. If *padAj* is NULL, the solver will set the initial values.

Remarks:

- paiRows should be sorted in ascending order.

LSaddNLPobj()

Description:

Adds NLP elements to the objective function for the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSaddNLPobj (pLSmodel pModel, int nCols, int *paiCols, double *padColj)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nCols	The total number of variables for which NLP elements will be added.
paiCols	A pointer to an integer vector containing the variable indices of the nonlinear elements.
padColj	A pointer to a double vector containing the initial nonzero coefficients of the NLP elements. If *padColj* is NULL, the solver will set the initial values.

Remarks:

- paiCols should be sorted in ascending order.

LSdeleteCones()

Description:

Deletes a set of cones in the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

Int	LSdeleteCones(pLSmodel pModel, int nCones, int *paiCones)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nCones	The number of cones in the model to delete.
paiCones	A pointer to a vector containing the indices of the cones that are to be deleted.

LSdeleteConstraints()

Description:

Deletes a set of constraints in the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSdeleteConstraints(pLSmodel pModel, int nCons, int *paiCons)

Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nCons	The number of constraints in the model to delete.
paiCons	A pointer to a vector containing the indices of the constraints that are to be deleted.

LSdeleteQCterms()

Description:

Deletes the quadratic terms from a set of constraints in the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSdeleteQCterms(pLSmodel pModel, int nCons, int *paiCons)

Input Arguments:

Name	Description
pModel	An instance of *LSmodel*.
nCons	The number of constraints in the model whose quadratic terms will be deleted.
paiCons	A pointer to a vector containing the indices of the constraints whose quadratic terms will be deleted.

LSdeleteNLPobj()

Description:

Deletes NLP elements from the objective function for the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

Int	LSdeleteNLPobj (pLSmodel pModel, int nCols, int *paiCols)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nCols	The number of variables for which NLP elements will be deleted.
paiCols	A pointer to a vector containing the indices of the variables whose NLP elements are to be deleted.

LSdeleteAj()

Description:

Deletes all the elements in the specified column for the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

Int	LSdeleteAj (pLSmodel pModel, int iVar1, int nRows, int *paiRows)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
iVar1	The index of the variable whose elements will be deleted.
nRows	The number of constraints at which elements will be deleted.
paiRows	A pointer to an integer vector containing the row indices of the elements to be deleted. The indices are required to be in ascending order.

LSdeleteSemiContVars()

Description:

Deletes a set of semi-continuous variables in the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSdeleteSemiContVars(pLSmodel pModel, int nSC, int *SCndx)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nSC	The number of semi-continuous variables in the model to delete.
SCndx	A pointer to a vector containing the indices of the semi-continuous variables that are to be deleted.

LSdeleteSETS()

Description:

Deletes the sets in the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSdeleteSETS(pLSmodel pModel, int nSETS, int *SETSndx)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nSETS	The number of sets in the model to delete.
SETSndx	A pointer to a vector containing the indices of the sets that are to be deleted.

LSdeleteVariables()

Description:

Deletes a set of variables in the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSdeleteVariables(pLSmodel pModel, int nVars, int *paiVars)

Input Arguments:

Name	Description
pModel	An instance of *LSmodel*.
nVars	The number of variables in the model to delete.
paiVars	A pointer to a vector containing the indices of the variables that are to be deleted.

LSmodifyAj()

Description:

Modifies the coefficients for a given column at specified constraints.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSmodifyAj(pLSmodel pModel, int iVar1, int nRows, int * paiCons, double *padAj)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
iVar1	The index of the variable to modify the constraint coefficients.
nCons	Number of constraints to modify.
paiCons	A pointer to an array of the indices of the constraints to modify.
padAj	A pointer to a double precision array containing the values of the new coefficients.

LSmodifyCone()

Description:

Modifies the data for the specified cone.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSmodifyCone(pLSmodel pModel, char cConeType, int iConeNum, int iConeNnz, int *paiConeCols)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
cConeType	A character variable specifying the new type of the cone.
iConeNum	An integer scalar that refers to the index of the cone to modify.
iConeNnz	An integer scalar that refers to the number of variables characterizing the cone.
paiConeCols	An integer vector that keeps the indices of the variables characterizing the cone. Its size should be *iConeNnz*.

LSmodifyConstraintType()

Description:

Modifies the type or direction of a set of constraints.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSmodifyConstraintType(pLSmodel pModel, int nCons, int *paiCons, char *pachContypes)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nCons	Number of constraints to modify.
paiCons	A pointer to an array of the indices of the constraints to modify.
pachContypes	A pointer to a character vector in which each element is either: 'L', 'E', 'G' or 'N' indicating each constraint's type.

Remarks:

- A constraint can be disabled by making its type 'N'.
- To modify the direction of the objective, use the function *LSsetModIntParameter* (model, LS_IPARAM_OBJSENSE, value), where value is either LS_MIN or LS_MAX.

LSmodifyObjConstant()

Description:

Modifies the objective's constant term for a specified model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSmodifyObjConstant(pLSmodel pModel, double dObjconst)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
dObjconst	The new objective constant term.

Remarks:

- To modify the objective's coefficients, see *LSmodifyObjective*().

LSmodifyLowerBounds()

Description:

Modifies selected lower bounds in a given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSmodifyLowerBounds(pLSmodel pModel, int nVars, int *paiVars, double *padL)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nVars	The number of bounds in the model to modify.
paiVars	A pointer to an integer vector containing the indices of the variables for which to modify the lower bounds.
padL	A pointer to a double precision vector containing the new values of the lower bounds on the variables.

LSmodifyObjConstant()

Description:

Modifies the objective's constant term for a specified model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSmodifyObjConstant(pLSmodel pModel, double dObjconst)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
dObjconst	The new objective constant term.

Remarks:

- To modify the objective's coefficients, see *LSmodifyObjective*().

LSmodifyObjective()

Description:

Modifies selected objective coefficients of a given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSmodifyObjective(pLSmodel pModel, int nVars, int *paiVars, double *padC)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nVars	Number of objective coefficients to modify.
paiVars	A pointer to an integer vector containing a list of the indices of the objective coefficients to modify.
padC	A pointer to a double precision vector containing the new values for the modified objective coefficients.

Remarks:

- To modify the objective's constant term, see *LSmodifyObjConstant*().

LSmodifyRHS()

Description:

Modifies selected constraint right-hand sides of a given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSmodifyRHS(pLSmodel pModel, int nCons, int *paiCons, double *padB)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nCons	The number of constraint right-hand sides to modify.
paiCons	A pointer to an integer vector containing the indices of the constraints whose right-hand sides are to be modified.
padB	A pointer to a double precision vector containing the new right-hand side values for the modified right-hand sides.

LSmodifySemiContVars()

Description:

Modifies data of a set of semi-continuous variables in the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSmodifySemiContVars(pLSmodel pModel, char nSC, int *piVarndx, double *padl, double *padu)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nSC	The number of semi-continuous variables to modify.
piVarndx	A pointer to an integer vector containing the indices of the variables whose data are to be modified.
padl	A pointer to a double precision vector containing the new lower bound values for the semi-continuous variables.
padu	A pointer to a double precision vector containing the new upper bound values for the semi-continuous variables.

LSmodifySET()

Description:

Modifies set data in the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSmodifySET(pLSmodel pModel, char cSETtype, int iSETnum, int iSETnnz, int *paiSETcols)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
cSETtype	A character variable containing the new type for the specified set.
iSETnum	An integer variable containing the index of the set to apply the modification.
iSETnnz	An integer variable containing the number of variables in the set specified with *iSETnum*.

paiSETcols	A pointer to an integer array containing the indices of variables in the set specified with *iSETnum*.

LSmodifyUpperBounds()

Description:

Modifies selected upper bounds in a given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSmodifyUpperBounds(pLSmodel pModel, int nVars, int *paiVars, double *padU)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nVars	The number of bounds in the model to modify.
paiVars	A pointer to an integer vector containing the indices of the variables for which to modify the upper bounds.
padU	A pointer to a double precision vector containing the new values of the upper bounds.

LSmodifyVariableType()

Description:

Modifies the types of the variables of the given model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSmodifyVariableType(pLSmodel pModel, int nVars, int *paiVars, char *pachVartypes)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nVars	Number of variables to modify.
paiVars	A pointer to an array of the indices of the variables to modify.
pachVartypes	A pointer to a character vector containing the types of variables. Valid values for each variable are 'C', 'B', or 'I' for continuous, binary, or general integer, respectively.

Remarks:

- To modify the direction of the objective, use the function *LSsetModelIntParameter*(model, LS_IPARAM_OBJSENSE, value), where value is either LS_MIN or LS_MAX.

Model and Solution Analysis Routines

The routines in this section allow you to analyze models and their solutions, such as performing sensitivity analysis of optimal solutions or debugging infeasible or unbounded linear programs. For a more detailed overview, see Chapter 10, *Analyzing Models and Solutions*.

LSfindBlockStructure

Description:

Examines the nonzero structure of the constraint matrix and tries to identify block structures in the model. If neither linking rows nor linking columns exist, then the model is called "totally decomposable". Unless total decomposition is requested, the user should specify as an input the number of blocks to decompose the matrix into.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSfindBlockStructure(pLSmodel pModel, int nBlock, int nType)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nBlock	An integer indicating the number of blocks to decompose the coefficient matrix into. The value of this argument is ignored if total decomposition is requested.
nType	An integer scalar indicating the type of decomposition requested. The possible values are identified with the following macros: LS_LINK_BLOCKS_NONE: Try total decomposition (no linking rows or columns). LS_LINK_BLOCKS_COLS: The decomposed model will have dual angular structure (linking columns). LS_LINK_BLOCKS_ROWS: The decomposed model will have block angular structure (linking rows). LS_LINK_BLOCKS_BOTH: The decomposed model will have both dual and block angular structure (linking rows and columns). LS_LINK_BLOCKS_FREE: Solver decides which type of decomposition to use.

Remarks:

- Only one stage of decomposition is attempted (i.e., no attempt is made to find further decomposition within a sub-block).
- The block structure obtained can be accessed by *LSgetBlockStructure()*.
- Refer to Chapter 10, *Analyzing Models*, for details on block structures.

LSfindIIS()

Description:

Finds an irreducibly inconsistent set (IIS) of constraints for an infeasible model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSfindIIS(pLSmodel pModel, int nLevel)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nLevel	An integer indicating the level of analysis in finding the IIS. Bit mask values are: LS_NECESSARY_ROWS = 1, LS_NECESSARY_COLS = 2, LS_SUFFICIENT_ROWS = 4, LS_SUFFICIENT_COLS = 8.

Remarks:

- The IIS obtained can be accessed by *LSgetIIS()*.
- Refer to Chapter 10, *Analyzing Models*, for details on debugging a model.

LSfindIUS()

Description:

Finds an irreducibly unbounded set (IUS) of columns for an unbounded linear program.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

Int	LSfindIUS(pLSmodel pModel, int nLevel)

Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
nLevel	An integer indicating the level of detail of the analysis in finding the IUS. Significant bit mask values are:

	LS_NECESSARY_COLS = 2, LS_SUFFICIENT_COLS = 8.

Remarks:

- The IUS obtained, can be accessed by *LSgetIUS()*.
- Refer to Chapter 10, *Analyzing Models*, for details on debugging a model.

LSgetBestBounds()

Description:

Finds the best implied variable bounds for the specified model by improving the original bounds using extensive preprocessing and probing.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetBestBounds(pLSmodel pModel, double *padBestL, double *padBestU)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padBestL	A double precision vector containing the best implied lower bounds if different from NULL. This vector must at least have as many entries as the number of variables in the model.
padBestU	A double precision vector containing the best implied upper bounds if different from NULL. This vector must at least have as many entries as the number of variables in the model.

LSgetBlockStructure()

Description:

Retrieves the block structure information following a call to *LSfindBlockStructure*.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*.

Prototype:

int	LSgetBlockStructure(pLSmodel pModel, int *pnBlock, int *panRblock, int *panCblock, int *pnType)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
pnBlock	A pointer to an integer scalar that contains the number of blocks to decompose the model matrix into. If *nType* = LS_LINK_BLOCKS_NONE, then **pnBlock* functions as an output argument, which will contain the number of independent blocks identified (provided that total decomposition is successful). Otherwise, it serves as an input argument where the solver attempts to decompose the model into **pnBlock* blocks linked by a set of rows or/and columns.
panRblock	A pointer to an integer vector in which information about the block membership of the constraints is to be placed. The length of this vector must be ≥ the number of constraints in the model. The *i*-th element of this array returns information on the *i*-th constraint as follows: 0: The row is a member of the linking (row) block. *k*>0: The row is a member of the *k*-th block. where 1 <= *k* <= **pnBlock*.
panCblock	A pointer to an integer vector in which information about the block membership of the variables is to be placed. The length of this vector must be ≥ the number of variables in the model. The *j*-th element of this array contains information on the *j*-th column as follows: 0: The column is a member of the linking (column) block. *k*>0: The column is a member of the *k*-th block. where 1 <= *k* <= **pnBlock*.
pnType	A pointer to an integer returning the type of the decomposition. The following macros identify possible values: LS_LINK_BLOCKS_NONE: Try total decomposition (no linking rows or columns).

	LS_LINK_BLOCKS_COLS: The decomposed model will have dual angular structure (linking columns). LS_LINK_BLOCKS_ROWS: The decomposed model will have block angular structure (linking rows). LS_LINK_BLOCKS_BOTH: The decomposed model will have both dual and block angular structure (linking rows and columns). LS_LINK_BLOCKS_FREE: Solver decides which type of decomposition to use.

Remarks:

- For more information on decomposition and linking structures, refer to Chapter 10, *Analyzing Models*.

LSgetBoundRanges()

Description:

Retrieves the maximum allowable decrease and increase in the primal variables for which the optimal basis remains unchanged.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetBoundRanges (pLSmodel pModel, double *padDec, double *padInc)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padDec	A pointer to a double precision vector that keeps the maximum allowable decrease in the lower and upper bounds. The vector size should be greater-than-or-equal-to the number of variables.
padInc	A pointer to a double precision vector that keeps the maximum allowable increase in the lower and upper bounds. The vector size should be greater-than-or-equal-to the number of variables.

LSgetConstraintRanges()

Description:

Retrieves the maximum allowable decrease and increase in the right-hand side values of constraints for which the optimal basis remains unchanged.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetConstraintRanges (pLSmodel pModel, double *padDec, double *padInc)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padDec	A pointer to a double precision vector that keeps the maximum allowable decrease in the right-hand sides of constraints. The size of this vector should be greater-than-or-equal-to the number of constraints.
padInc	A pointer to a double precision vector that keeps the maximum allowable increase in the right-hand sides of constraints. The size of this vector should be greater-than-or-equal-to the number of constraints.

LSgetIIS()

Description:

Retrieves the irreducibly inconsistent set (IIS) of constraints for an infeasible model following a call to *LSfindIIS*(). Any of the pointers to the names can be input as NULL if the corresponding information is not required.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetIIS(pLSmodel pModel, int *pnSuf_r, int *pnIIS_r, int *paiCons, int *pnSuf_c, int *pnIIS_c, int *paiVars, int *panBnds)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
pnSuf_r	A pointer to the number of constraints in the *sufficient* set.
pnIIS_r	A pointer to the number of rows in the IIS.
paiCons	A pointer to a vector of size **pnIIS_r* containing the indices of the rows in the IIS. The locations *paiCons*[0] to *paiCons*[**pnSuf_r* –1] keep the indices of the sufficient rows.
pnSuf_c	A pointer to the number of column bounds in the *sufficient* set.
pnIIS_c	A pointer to the number of column bounds in the IIS.
paiVars	A pointer to a vector of size **pnIIS_c* containing the indices of the column bounds in the IIS. The locations *paiVars*[0] to *paiVars*[**pnSuf_c* –1] store the indices of the members of the sufficient column bounds.
panBnds	A pointer to a vector of size **pnIIS_c* indicating whether the lower or the upper bound of the variable is in the IIS. Its elements are –1 for lower bounds and +1 for upper bounds.

Remarks:

- This tool assumes that the user has recently attempted optimization on the model and the solver returned a basic, but infeasible, solution. If an infeasible basis is not resident in the solver, the diagnostic tool cannot initiate the processes to isolate an IIS. Cases that violate this condition are: the pre-solver has detected the infeasibility of the model, or the barrier solver has terminated without performing a basis crossover. To obtain an IIS for such cases, the pre-solve option should be turned off and the model must be optimized again.
- Refer to Chapter 10, *Analyzing Models*, for details on debugging a model.

LSgetIUS()

Description:

Retrieves the irreducibly unbounded set (IUS) of columns for an unbounded linear program following a call to *LSfindIUS*(). Any of the pointers to the names can be input as NULL if the corresponding information is not required

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetIUS(pLSmodel pModel, int *pnSuf, int *pnIUS, int *paiVars)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
pnSuf	A pointer to the number of columns in the *sufficient* set.
pnIUS	A pointer to the number of columns in the IUS.
paiVars	A pointer to a vector of size **pnIUS* containing the indices of the columns in the IUS. The locations *paiVars*[0] to *paiVars*[**pnSuf* –1] store the indices of the members of the sufficient set.

Remarks:

- Refer to Chapter 10, *Analyzing Models*, for details on debugging a model.

LSgetObjectiveRanges()

Description:

Retrieves the maximum allowable decrease and increase in objective function coefficients for which the optimal basis remains unchanged.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetObjectiveRanges(pLSmodel pModel, double *padDec, double *padInc)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
padDec	A pointer to a double precision vector that keeps the maximum allowable decrease in the objective function coefficients. The size of this vector should be greater-than-or-equal-to the number of variables.
PadInc	A pointer to a double precision vector that keeps the maximum allowable increase in the objective function coefficients. The vector size should be greater-than-or-equal-to the number of variables.

Error Handling Routines

The routines in this section allow you to get detailed information about the errors that occur during calls to LINDO API routines and while accessing a text file for I/O.

LSgetErrorMessage()

Description:

Retrieves the error message associated with the given error code.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetErrorMessage(pLSenv pEnv, int nErrorcode, char *pszMessage)

Input Arguments:

Name	Description
pEnv	A pointer to an instance of *LSenv*. Error messages are stored in this environment.
nErrorcode	An integer referring to the error code.

Output Arguments:

Name	Description
pszMessage	The error message associated with the given error code. It is assumed that memory has been allocated for this string.

Remarks:

- The length of the longest message will not exceed LS_MAX_ERROR_MESSAGE_LENGTH, including the terminating null character. So, be sure to allocate at least this many bytes before calling *LSgetErrorMessage*().

LSgetErrorRowIndex()

Description:

Retrieves the index of the row where a numeric error has occurred.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	(pLSmodel pModel, int *piRow);

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
piRow	A pointer to an integer variable to return the row index with numeric error.

LSgetFileError()

Description:

Provides the line number and text of the line in which an error occurred while reading or writing a file.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetFileError (pLSmodel pModel, int *pnLinenum, char *pszLinetxt)

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.

Output Arguments:

Name	Description
pnLinenum	A pointer to an integer that returns the line number in the I/O file where the error has occurred.
pszLinetxt	A pointer to a null terminated string that returns the text of the line where the error has occurred.

Advanced Routines

The routines in this section perform specialized functions. Users interested in only building and solving a model will not need to access the routines detailed in this section. Users who are developing customized solution procedures, however, may find these routines useful.

LSdoBTRAN()

Description:

Does a so-called backward transformation. That is, the function solves the linear system $B^TX = Y$, where B^T is the transpose of the current basis of the given linear program and Y is a user specified vector.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSdoBTRAN(pLSmodel pModel, int *pcYnz, int *paiY, double *padY, int *pcXnz, int *paiX, double *padX)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
pcYnz	A pointer to an integer containing the number of nonzeros in the right-hand side vector *Y*.
paiY	A pointer to an integer vector containing the positions of the nonzeros in *Y*.
padY	A pointer to a double precision vector containing the coefficients of the nonzeros in *Y*.

Output Arguments:

Name	Description
pcXnz	A pointer to an integer containing the number of nonzeros in the solution vector *X*.
paiX	A pointer to an integer vector containing the positions of the nonzeros in *X*. You must allocate the memory for this vector, and should allocate at least *m* elements, where *m* is the number of constraints.
padX	A pointer to a double precision vector containing the coefficients of the nonzeros in *X*. You must allocate the memory for this vector, and should allocate at least *m* elements, where *m* is the number of constraints.

Remarks:

- This routine should be called only after optimizing the model.

LSdoFTRAN()

Description:

Does a so-called forward transformation. That is, the function solves the linear system $B X = Y$, where B is the current basis of the given linear program, and Y is a user specified vector.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSdoFTRAN(pLSmodel pModel, int *pcYnz, int *paiY, double *padY, int *pcXnz, int *paiX, double *padX)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
pcYnz	A pointer to an integer containing the number of nonzeros in the right-hand side vector *Y*.
paiY	A pointer to an integer vector containing the positions of the nonzeros in *Y*.
padY	A pointer to a double precision vector containing the coefficients of the nonzeros in *Y*.

Output Arguments:

Name	Description
pcXnz	A pointer to an integer containing the number of nonzeros in the solution vector, *X*.
paiX	A pointer to a vector containing the positions of the nonzeros in *X*.
padX	A pointer to a double precision vector containing the coefficients of the nonzeros in *X*.

Remarks:

- This routine should be called only after optimizing the model.

LScalcConFunc()

Description:

Calculates the constraint activity at a primal solution. The specified model should be loaded by using *LSloadInstruct()*.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*.

Prototype:

int	LScalcConFunc(pLSmodel pModel, int iCon, double *padPrimal, double *pdValue,)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
iCon	An integer containing the index of the constraint whose activity is requested.
padPrimal	A pointer to a double precision vector that contains the primal solution at which the constraint activity will be computed.

Output Arguments:

Name	Description
pdValue	A double precision variable that returns the constraint activity at the given primal solution *padPrimal*.

LScalcConGrad()

Description:

Calculates the partial derivatives of the function representing a constraint with respect to a set of primal variables. The specified model should be loaded by using *LSloadInstruct()*.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LScalcConGrad(pLSmodel pModel, int iCon, double *padPrimal, int nVar, int *paiVar, double *padVar)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
iCon	An integer containing the index of the constraint whose partial derivatives is requested.
padPrimal	A pointer to a double precision vector that contains the primal solution at which the partial derivatives of the constraint will be evaluated.
nVar	An integer scalar indicating the number of variables to compute the partial derivatives for.
paiVar	A pointer to an integer vector that contains the indices of the variables to compute the partial derivatives for.

Output Arguments:

Name	Description
padVar	A pointer to a double precision vector that returns the partial derivatives of the variables indicated by *paiVar*[].

LScalcObjFunc()

Description:

Calculates the objective function value at a primal solution. The specified model should be loaded by using *LSloadInstruct()*.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LScalcObjFunc(pLSmodel pModel, double *padPrimal , double *pdPobjval,)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
padPrimal	A pointer to a double precision vector that contains the primal solution at which the objective function will be evaluated.

Output Arguments:

Name	Description
pdPobjval	A double precision variable that returns the objective value for the given primal solution.

LScalcObjGrad()

Description:

Calculates the partial derivatives of the objective function with respect to a set of primal variables. The specified model should be loaded by using *LSloadInstruct()*.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LScalcObjGrad(pLSmodel pModel, double *padPrimal, int nVar, int *paiVar, double *padVar)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
padPrimal	A pointer to a double precision vector that contains the primal solution at which the partial derivatives of the objective function will be evaluated.
nVar	An integer scalar indicating the number of variables to compute the partial derivatives for.
paiVar	A pointer to an integer vector that contains the indices of the variables to compute the partial derivatives for.

Output Arguments:

Name	Description
padVar	A pointer to a double precision vector that returns the partial derivatives of the variables indicated by *paiVar*[].

Callback Management Routines

The routines in this section allow the user to set callback functions and manage callback information. Refer to Chapter 9, *Using Callback Functions*, for examples of using callback management routines.

LSgetCallbackInfo()

Description:

Returns information about the current state of the LINDO API solver during model optimization. This routine is to be called from your user supplied callback function that was set with *LSsetCallback*().

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetCallbackInfo(pLSmodel pModel, int nLocation, int nQuery, void *pvValue)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*. This should be the same instance as was passed to your user callback function from the LINDO API solver.
nLocation	The solver's current location. This parameter is passed to your callback function by the LINDO API solver.
nQuery	The information desired from LINDO API. Only the following select information can be obtained from the callback function: • LS_IINFO_SIM_ITER: Number of simplex iterations performed for solving a continuous problem. • LS_IINFO_BAR_ITER: Number of barrier iterations performed for solving a continuous problem. • LS_IINFO_NLP_ITER: Number of nonlinear iterations performed for solving a continuous problem. • LS_DINFO_POBJ: Primal objective value of a continuous problem. • LS_DINFO_DOBJ: Dual objective value of a continuous problem. • LS_DINFO_PINFEAS: Maximum primal infeasibility. • LS_DINFO_DINFEAS: Maximum dual infeasibility. • LS_DINFO_MSW_POBJ: Value of the incumbent objective value when using the multistart solver. • LS_IINFO_MSW_PASS: Number of multistart passes. • LS_IINFO_MSW_NSOL: Number of distinct

	solutions found when using the multistart solver. • LS_DINFO_MIP_OBJ: MIP objective value. • LS_DINFO_MIP_BESTBOUND: Best bound on MIP objective. • LS_IINFO_MIP_LPCOUNT: Number of LPs solved for solving a MIP. • LS_IINFO_MIP_BRANCHCOUNT: Number of branches generated for solving a MIP. • LS_IINFO_MIP_ACTIVENODES: Number of remaining nodes to be explored. • LS_IINFO_MIP_LTYPE: Type of the last MIP solution. • LS_IINFO_MIP_SIM_ITER: Number of simplex iterations performed for solving a MIP. • LS_IINFO_MIP_BAR_ITER: Number of barrier iterations performed for solving a MIP. • LS_IINFO_MIP_NLP_ITER: Number of nonlinear iterations performed for solving a MIP. • LS_IINFO_MIP_NUM_TOTAL_CUTS: Number of total cuts generated. • LS_IINFO_MIP_GUB_COVER_CUTS: Number of GUB cover cuts generated. • LS_IINFO_MIP_FLOW_COVER_CUTS: Number of flow cover cuts generated. • LS_IINFO_MIP_LIFT_CUTS: Number of lifted knapsack covers generated. • LS_IINFO_MIP_PLAN_LOC_CUTS: Number of plant location cuts generated. • LS_IINFO_MIP_DISAGG_CUTS: Number of disaggregation cuts generated. • LS_IINFO_MIP_KNAPSUR_COVER_CUTS: Number of surrogate knapsack cover cuts generated. • LS_IINFO_MIP_LATTICE_CUTS: Number of lattice cuts generated. • LS_IINFO_MIP_GOMORY_CUTS: Number of Gomory cuts generated. • LS_IINFO_MIP_COEF_REDC_CUTS: Number of coefficient reduction cuts generated. • LS_IINFO_MIP_GCD_CUTS: Number of GCD cuts generated. • LS_IINFO_MIP_OBJ_CUT: Number of objective cuts generated. • LS_IINFO_MIP_BASIS_CUTS: Number of basis cuts generated. • LS_IINFO_MIP_CARDGUB_CUTS: Number of cardinality/GUB cuts generated. • LS_IINFO_MIP_CONTRA_CUTS: Number of contra cuts generated.

	• LS_IINFO_MIP_CLIQUE_CUTS: Number of clique cuts generated. • LS_DINFO_GOP_OBJ: Objective value of the global optimal solution of a GOP. • LS_DINFO_GOP_BESTBOUND: Best bound on the objective value of a GOP. • LS_IINFO_GOP_STATUS: Solution status of a GOP. • LS_IINFO_GOP_LPCOUNT: Number of LPs solved for solving a GOP. • LS_IINFO_GOP_NLPCOUNT: Number of NLPs solved for solving a GOP. • LS_IINFO_GOP_MIPCOUNT: Number of MIPs solved for solving a GOP. • LS_IINFO_GOP_NEWSOL: If a new GOP solution has been found or not. • LS_IINFO_GOP_BOX: Number of explored boxes • LS_IINFO_GOP_BBITER: Number of iterations performed during a major GOP iteration. • LS_IINFO_GOP_SUBITER: Number of iterations performed during a minor GOP iteration. • LS_IINFO_GOP_ACTIVEBOXES: Number of active boxes at current state for solving a GOP. • LS_IINFO_GOP_MIPBRANCH: Number of branches created for solving a GOP.

Output Arguments:

Name	Description
pvValue	This is a pointer to a memory location where LINDO API will return the requested information. You must allocate sufficient memory for the requested information prior to calling this function.

Remarks:

- *LSgetInfo*() cannot be used during callbacks.
- Query values whose names begin with LS_IINFO return integer values, while those whose names begin with LS_DINFO return double precision floating point values.
- Refer to Chapter 9, *Using Callback Functions,* for additional information.

LSgetMIPCallbackInfo()

Description:

Returns information about the current state of the LINDO API branch-and-bound solver. This routine is to be called from your user supplied callback functions that were established with the calls *LSsetCallback*()and *LSsetMIPCallback*().

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSgetMIPCallbackInfo(pLSmodel pModel, int nQuery, void *pvValue)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*. This should be the same instance as was passed to your user callback function from the LINDO API solver.
nQuery	This is the information desired from LINDO API. All information that can be accessed via *LsgetCallbackInfo*() is available.

Output Arguments:

Name	Description
pvValue	This is a pointer to a memory location where LINDO API will return the requested information. You must allocate sufficient memory for the requested information prior to calling this function.

Remarks:

- Query values whose names begin with LS_IINFO return integer values, while those values whose names begin with LS_DINFO return double precision floating point values.
- Refer to Chapter 9, *Using Callback Functions,* for additional information on the use of callback functions.

LSsetCallback()

Description:

Supplies LINDO API with the address of the callback function that will be called at various points throughout all components of LINDO API. The user supplied callback function can be used to report the progress of the solver routines to a user interface, interrupt the solver, etc.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSsetCallback(pLSmodel pModel, int (CALLBACKTYPE *pcbFunc)(LSmodel*, int, void*), void *pvData)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
pcbFunc	A pointer to the user supplied callback function.
pvData	A pointer to any data you wish to access within the callback function. Of course, this can be a pointer to a structure, allowing any amount of information to be passed.

Remarks:

- To disable the callback function, call this routine with the callback function set to NULL.
- To control the frequency of callbacks, use *LSsetEnvDouParameter*() to set parameter LS_DPARAM_CALLBACKFREQ. This parameter is the number of seconds (approximately) between callbacks.
- If the value returned by the callback function is nonzero, the solver will interrupt and the control of the application program will pass to the user.
- Refer to the *lindo.h* file for CALLBACKTYPE macro definition.
- Refer to Chapter 9, *Using Callback Functions,* for additional information.

LSsetEnvLogFunc ()

Description:

Supplies the specified environment with the addresses of a) the *pLogfunc*() that will be called each time LINDO API logs message and b) the address of the user data area to be passed through to the *pUsercalc*() routine.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSsetEnvLogFunc (pLSenv pEnv, printLOG_t *pLogfunc, void *pUserData)

Input Arguments:

Name	Description
pEnv	A pointer to an instance of *LSenv*.
pLogfunc	A pointer to the subroutine that will be called to log messages.
pUserData	A pointer to a "pass through" data area in which your calling application may place information about the functions to be calculated. Whenever LINDO API calls your subroutine *pUsercalc*(), it passes through the pointer *pUserData* which could contain data to be used in the computation of the final value. Passing data in this manner will ensure that your application remains thread safe.

LSsetFuncalc ()

Description:

Supplies LINDO API with the addresses of a) the user-supplied function computing the routine *pFuncalc*() (see Chapter 7) that will be called each time LINDO API needs to compute a row value, and b) the address of the user data area to be passed through to the *pFuncalc*() routine.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSsetFuncalc (pLSmodel pModel, Funcalc_type *pFuncalc, void *pUserData)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
pFuncalc	A pointer to the subroutine that computes the value of a specified nonlinear row. See the definition of *pFuncalc*() in Chapter 7, *Solving Nonlinear Programs*, for details on this function's prototype.
pUserData	A pointer to a "pass through" data area in which your calling application may place information about the functions to be calculated. Whenever LINDO API calls your subroutine *pFuncalc*(), it passes through the pointer *pUserData*. All data that *pFuncalc*() needs to compute function values should be in the *pUserData* memory block. Passing data in this manner will ensure that your application remains thread safe.

Remarks:

- Visual Basic users can use the AddressOf operator to pass the address of a routine to *LSsetFuncalc*(). The supplied routine must be in a VB module, or the AddressOf operator will fail.

LSsetGradcalc()

Description:

Supplies LINDO API with the addresses of a) the *pGradcalc* () callback function (see Chapter 7, *Solving Nonlinear Programs*) that will be called each time LINDO API needs a gradient (i.e., vector of partial derivatives), and b) the data area to be passed through to the gradient computing routine. This data area may be the same one supplied to *LSsetFuncalc*().

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSsetGradcalc (pLSmodel pModel, Gradcalc_type *pGradcalc, void *pUserData, int nLenUseGrad, int *pnUseGrad);

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
pGradcalc	A pointer to the subroutine that computes the gradients for specified nonlinear rows. See the definition of *pGradcalc* () in Chapter 7, *Solving Nonlinear Programs*, for details on this function's interface.
pUserData	A pointer to a "pass through" data area in which your calling application may place information about the partial derivatives to be calculated. Whenever LINDO API calls your subroutine *pGradcalc*(), it passes through the pointer *pUserData*. All data that *pGradcalc*() needs to compute gradients should be in the *pUserData* memory block. Passing data in this manner will ensure that your application remains thread safe.
nLenUseGrad	An integer indicating how many nonlinear rows will make use of the *pGradcalc*() routine. 0 is interpreted as meaning that no functions use a *pGradcalc*() routine, thus meaning that partials on all functions are computed with finite differences. A value of -1 is interpreted as meaning the partials on all nonlinear rows will be computed through the *pGradcalc*() routine. A value greater than 0 and less-than-or-equal-to the number of nonlinear rows is interpreted as being the number of nonlinear rows that make use of the *pGradcalc* () routine. And, the list of indices of the rows that do so is contained in the following array, *pnUseGrad*.
pnUseGrad	An integer array containing the list of nonlinear rows that make use of the *pGradcalc*() routine. You should set this pointer to NULL if *nLenUseGrad* is 0 or -1. Otherwise, it should point to an array of dimension *nLenUseGrad*, where *pnUseGrad*[*j*] is the index of the *j*-th row whose partial derivatives are supplied through the *pGradcalc*() function. A value of -1 indicates the

	objective row.

Remarks:

- *LSsetGradcalc*() need not be called. In that case, gradients will be approximated by finite differences.
- Visual Basic users can use the *AddressOf* operator to pass the address of a routine to *LSsetGradcalc*(). The supplied routine must be in a VB module, or the *AddressOf* operator will fail.

LSsetMIPCallback()

Description:

Supplies LINDO API with the address of the callback function that will be called each time a new integer solution has been found to a mixed-integer model.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSsetMIPCallback(pLSmodel pModel, int (CALLBACKTYPE *pMIP_caller)(LSmodel*, void*, double, double*), void *pvData)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
pMIP_caller	A pointer to your user supplied callback function.
pvData	A pointer to any data you wish to access within the callback function. Of course, this can be a pointer to a structure, allowing any amount of information to be passed.

Remarks:

- To disable the MIP callback function, call this routine with the callback function set to NULL.
- To retrieve information in your MIP callback routine, see *LSgetMIPCallbackInfo*().
- To interrupt the mixed-integer optimizer before a new integer solution is found or in between new integer solutions, set a general callback function via *LSsetCallback*().
- Refer to the *lindo.h* file for the CALLBACKTYPE macro definition.
- Refer to Chapter 9, *Callback Functions,* for additional information.

LSsetModelLogFunc ()

Description:

Supplies the specified model with the addresses of a) the *pLogfunc* () that will be called each time LINDO API logs message and b) the address of the user data area to be passed through to the *pUsercalc*() routine.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSsetModelLogFunc (pLSmodel pModel, printLOG_t *pLogfunc, void *pUserData)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
pLogfunc	A pointer to the subroutine that will be called to log messages.
pUserData	A pointer to a "pass through" data area in which your calling application may place information about the functions to be calculated. Whenever LINDO API calls your subroutine *pUsercalc*(), it passes through the pointer *pUserData* which could contain data to be used in the computation of the final value. Passing data in this manner will ensure that your application remains thread safe.

LSsetUsercalc ()

Description:

Supplies LINDO API with the addresses of a) the *pUsercalc*() (see Chapter 7) that will be called each time LINDO API needs to compute the value of the user-defined function and b) the address of the user data area to be passed through to the *pUsercalc*() routine.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSsetFuncalc (pLSmodel pModel, user_callback_t *pUsercalc, void *pUserData)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.
pUsercalc	A pointer to the subroutine that computes the value of a user-defined function. See the definition of *pUsercalc*() in Chapter 7, *Solving Nonlinear Programs*, for details on this function's prototype.
pUserData	A pointer to a "pass through" data area in which your calling application may place information about the functions to be calculated. Whenever LINDO API calls your subroutine *pUsercalc*(), it passes through the pointer *pUserData* which could contain data to be used in the computation of the final value. Passing data in this manner will ensure that your application remains thread safe.

Remarks:

- *LSsetGradcalc*() need not be called. In that case, gradients will be approximated by finite differences.

Memory Management Routines

The routines in this section allow the user to manage the memory used by the LINDO API solvers.

LSfreeGOPSolutionMemory()

Description:

This routine frees up the arrays associated with the GOP solution of a given model. After freeing the memory, you will lose all access to the information associated to GOP solutions.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSfreeGOPSolutionMemory(pLSmodel pModel)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

LSfreeHashMemory()

Description:

This routine frees up work arrays associated with a given model's variable name hashing. This will release memory to the system pool, but will cause any subsequent variable name lookup to pause to regenerate these tables.

Returns:

if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

void	LSfreeHashMemory(pLSmodel pModel)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

Remarks:

- A model also stores work arrays for the solver. These arrays may be freed by a call to *LSfreeSolverMemory*().

LSfreeMIPSolutionMemory()

Description:

This routine frees up the arrays associated with the MIP solution of a given model. After freeing the memory, you will lose all access to the information associated to MIP solutions.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSfreeMIPSolutionMemory(pLSmodel pModel)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

LSfreeSolutionMemory()

Description:

This routine frees up the arrays associated with the solution of a given model. This will release the associated memory blocks to the system, but will not cause the solver to loose any warm start capability for the model on its next run. However, you will lose all access to the model's solution information.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

int	LSfreeSolutionMemory(pLSmodel pModel)

Input Arguments:

Name	Description
pModel	A pointer to an instance of *LSmodel*.

LSfreeSolverMemory()

Description:

This routine frees up solver work arrays associated with a given model. This will release the associated memory to the system, but will cause any subsequent reoptimization of the model to take more time. In other words, the solver will lose its warm start capability for the model on its next run. Note that by freeing solver memory, you will *not* lose access to the model's solution information.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes.*

Prototype:

void	LSfreeSolverMemory(pLSmodel pModel)

Input Arguments:

Name	Description
pModel	A pointer to an instance of LSmodel.

Remarks:

- A model also stores work arrays for variable name hashing. These arrays may be freed by a call to *LSfreeHashMemory*().

Random Number Generation Routines

Random Number Generator Functions.

LScreateRG ()

Description:

Create a new random generator object.

Returns:

pRG A reference to a random number generator.

Prototype:

pLSrandGen	LScreateRG (pLSenv pEnv, int nMethod)

Input Arguments:

Name	Description
pEnv	A reference to an instance of LSenv.
nMethod	An integer specifying the random number generator to use. Possible values are: • LS_RANDGEN_FREE • LS_RANDGEN_SYSTEM • LS_RANDGEN_LINDO1 • LS_RANDGEN_LINDO2 • LS_RANDGEN_LIN1 • LS_RANDGEN_MULT1 • LS_RANDGEN_MERSENNE

LSgetDoubleRV ()

Description:

Get the next standard uniform random variate in the stream.

Prototype:

double	LSgetDoubleRV (pLSrandGen pRG)

Input Arguments:

Name	Description
pRG	A reference to the random number generator.

LSgetDistrRV ()

Description:

Get the next double random variate of underlying distribution.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSgetDistrRV (pLSrandGen pRG, void * dResult)

Input Arguments:

Name	Description
pRG	A reference to the random number generator.
dResult	The next random value from underlying distribution

LSgetInitSeed ()

Description:

Get the seed initiated this random generator.

Prototype:

int	LSgetInitSeed (pLSrandGen pRG)

Input Arguments:

Name	Description
pRG	A reference to the random number generator.

LSgetInt32RV ()

Description:

Get the next integer random variate in the stream.

Prototype:

int	LSgetInt32RV (pLSrandGen pRG, int ib, int ie)

Input Arguments:

Name	Description
pRG	A reference to the random number generator.
ib	lower bound for random variate
ie	upper bound for random variate

LSsetRGSeed ()

Description:

Set an initialization seed for the random number generator.

Prototype:

void	LSsetRGSeed (pLSrandGen pRG, int seed)

Input Arguments:

Name	Description
pRG	A reference to the random number generator.
seed	An integer specifying the seed to set.

LSdisposeRG ()

Description:

Delete the specified random generator object.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

void	LSdisposeRG (pLSrandGen * *ppRG*)

Input Arguments:

Name	Description
ppRG	A reference to the address of a random number generator.

LSsetDistrRG ()

Description:

Set a cdfinv for the random generator.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsetDistrRG (pLSrandGen pRG, int nDistType)

Input Arguments:

Name	Description
pRG	A reference to the random number generator.
nDistType	An integer specifying the distribution type. See **LSsampCreate()** for possible values.

LSsetDistrParamRG ()

Description:

Set distribution parameters for internal cdfinv.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsetDistrParamRG (pLSrandGen pRG, int iParam, double dParam)

Input Arguments:

Name	Description
pRG	A reference to the random number generator.
iParam	A parameter index
dParam	A parameter value

Sampling Routines

Sampling Functions.

LSsampCreate ()

Description:

Create an instance of a sample (pLSsample) of specified distribution.

Returns:

A reference to an instance of **LSsample** object.

Prototype:

pLSsample	LSsampCreate (pLSenv pEnv, int nDistrType, int * perrorcode)

Input Arguments:

Name	Description
pEnv	A reference to an instance of LSenv object.
nDistrType	An integer specifying the distribution type. Possible values: one of the distribution functions listed in the table above *Distribution Function Macros*.
perrorcode	An reference to an integer returning the error code. See Appendix-A for possible values.

LSsampDelete ()

Description:

Delete the specified pLSsample object.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampDelete (pLSsample * pSample)

Input Arguments:

Name	Description
pSample	A reference to an instance of LSsample object.

LSsampLoadDiscretePdfTable ()

Description:

Load a PDF table for a user defined discrete distribution.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampLoadDiscretePdfTable (pLSsample pSample, int nLen, double * padProb, double * padVals)

Input Arguments:

Name	Description
pSample	A reference to an instance of LSsample object.
nLen	An integer specifying the table length.
padProb	A double array specifying the probabilities of outcomes.
padVals	A double array specifying the values of outcomes (optional)

Remarks:

- if nLen <=0, the table length will be set to default (100)

LSsampGetDiscretePdfTable ()

Description:

Get the PDF table from a discrete distribution sample.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampGetDiscretePdfTable (pLSsample pSample, int nLen, double * padProb, double * padVals)

Input Arguments:

Name	Description
pSample	A reference to an instance of LSsample object.
nLen	An integer to return the table length.
padProb	A double array to return the probabilities of outcomes.
padVals	A double array to return the values of outcomes (optional)

Remarks:

- Normally, this function should be called twice. The first call to get pnLen (with other arguments set to NULL) to allocate space for padProb and padVals. In the second call, padProb and padVals would be populated.

LSsampSetUserDistr ()

Description:

Specify a custom function to compute the PDF.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampSetUserDistr (pLSsample pSample, UserPdf * pFunc)

Input Arguments:

Name	Description
pSample	A reference to an instance of LSsample object.
pFunc	A user defined routine.

LSsampSetDistrParam ()

Description:

Set the specified parameter of the given distribution.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampSetDistrParam (pLSsample pSample, int iarg, double dargv)

Input Arguments:

Name	Description
pSample	A reference to an instance of LSsample object.
iarg	An integer specifying the index of the parameter.
dargv	A double precision value specifying the parameter value.

LSsampGetDistrParam ()

Description:

Get the specified parameter of a given distribution.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampGetDistrParam (pLSsample pSample, int iarg, double * dargv)

Input Arguments:

Name	Description
pSample	A reference to an instance of LSsample object.
iarg	An integer specifying the index of the parameter.
dargv	A double precision value specifying the parameter value.

LSsampEvalDistr ()

Description:

Evaluate the specified function associated with given distribution at specified point.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampEvalDistr (pLSsample pSample, int nFuncType, double dX, double * dResult)

Input Arguments:

Name	Description
pSample	A reference to an instance of LSsample object.
nFuncType	An integer specifying the function type to evaluate. Possible values are: • LS_PDF: probability density function. • LS_CDF: cummulative density function. • LS_CDFINV: inverse of cummulative density function. • LS_PDFDIFF: derivative of the probability density function.
dX	A double precision value to evaluate the specified function.
dResult	A reference to a double value to return the result.

LSsampSetRG ()

Description:

Set a random number generator object to the specified distribution.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampSetRG (pLSsample pSample, void * pRG)

Input Arguments:

Name	Description
pSample	A reference to an instance of LSsample object.
pRG	A reference to a random number generator.

LSsampGenerate ()

Description:

Generate a sample of size `nSampSize` with specified method from the underlying distribution.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampGenerate (pLSsample pSample, int nSampMethod, int nSampSize)

Input Arguments:

Name	Description
pSample	A reference to an instance of LSsample object.
nSampMethod	An integer specifying the sampling method. Possible values are: • LS_MONTECARLO • LS_LATINSQUARE (default) • LS_ANTITHETIC
nSampSize	An integer specifying the sample size. Possible values are nonnegative integers > 2.

LSsampGetPoints ()

Description:

Get a copy of the generated sample points.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampGetPoints (pLSsample pSample, int * pnSampSize, double * pX)

Input Arguments:

Name	Description
pSample	A reference to an instance of LSsample object.
pnSampSize	A reference to an integer specifying the sample size.
pX	A reference to a double pointer containing the sample.

Remarks:

Use LSdistGetSamplePtr for fast access to the sample.

LSsampGetPointsPtr ()

Description:

Get a reference to the generated sample points.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampGetPointsPtr (pLSsample pSample, int * pnSampSize, double ** pX)

Input Arguments:

Name	Description
pSample	A reference to an instance of LSsample object.
pnSampSize	A reference to an integer specifying the sample size.
pX	A reference to a double pointer containing the sample.

LSsampGetCIPoints ()

Description:

Get a copy of the correlation induced sample points.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampGetCIPoints (pLSsample pSample, int * pnSampSize, double *pX)

Input Arguments:

Name	Description
pSample	A reference to an instance of LSsample object.
pnSampSize	A refernce to an integer specifying the sample size.
pX	A reference to a double vector containing the sample.

LSsampGetCIPointsPtr ()

Description:

Get a reference to the correlation induced sample points.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampGetCIPointsPtr (pLSsample *pSample*, int * *pnSampSize*, double ** *pX*)

Input Arguments:

Name	Description
pSample	A reference to an instance of LSsample object.
pnSampSize	A reference to an integer specifying the sample size.
pX	A reference to a double pointer containing the sample.

LSsampGetCorrelationMatrix ()

Description:

Get the correlation structure between variables.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampGetCorrelationMatrix (pLSsample * paSample, int nDim, int iFlag, int nCorrType, int * QCnonzeros, int * QCvarndx1, int * QCvarndx2, double * QCcoef)

Input Arguments:

Name	Description
paSample	An array of instances of pLSsample
nDim	An integer specifying the number of variables (length of paSample)
iFlag	An integer specifying the sample (original or corr-induced). Possible values are: • 0 use independent sample • 1 use dependent (correlation induced) sample.
nCorrType	Correlation type. Possible values are: • LS_CORR_PEARSON (default) • LS_CORR_SPEARMAN • LS_CORR_KENDALL • LS_CORR_TARGET
QCnonzeros	A reference to an integer to return the number of nonzero correlation coefficients.
QCvarndx1	A vector containing the first index of variable the correlation term belongs to (QCnonzeros long)..
QCvarndx2	A vector containing the second index of variable the correlation term belongs to (QCnonzeros long)..
QCcoef	A vector containing the correlation terms (QCnonzeros long).

LSsampInduceCorrelation ()

Description:

Induce a target dependence structure between the stochastic elements via the specified correlation matrix. This matrix can be retrieved with LSgetCorrelationMatrix function with LS_CORR_TARGET as the argument.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampInduceCorrelation (pLSsample * paSample, int nDim, int nCorrType, int QCnonzeros, int * QCvarndx1, int * QCvarndx2, double * QCcoef)

Input Arguments:

Name	Description
paSample	An array of instances of LSsample
nDim	An integer specifying the number of variables (length of paSample)
nCorrType	Correlation type. Possible values are: • LS_CORR_PEARSON • LS_CORR_SPEARMAN • LS_CORR_KENDALL
QCnonzeros	The number of nonzero correlation coefficients.
QCvarndx1	A vector containing the first index of variable the correlation term belongs to (QCnonzeros long)..
QCvarndx2	A vector containing the second index of variable the correlation term belongs to (QCnonzeros long)..
QCcoef	A vector containing the correlation terms (QCnonzeros long).

Remarks:

Use LSdistGetSamplePtr for fast access to the sample.

LSsampGetInfo ()

Description:

Get information about the sample.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*

Prototype:

int	LSsampGetInfo (pLSsample pSample, int query, void * result)

Input Arguments:

Name	Description
paSample	An array of instances of LSsample
query	An integer specifying the information requested from the sample. Possible values are: • LS_IINFO_DIST_TYPE • LS_IINFO_SAMP_SIZE • LS_DINFO_SAMP_MEAN • LS_DINFO_SAMP_STD • LS_DINFO_SAMP_SKEWNESS • LS_DINFO_SAMP_KURTOSIS
result	A reference to the appropriate type to return the result.

Note:

Query values whose names begin with LS_IINFO take integer values while those whose names begin with LS_DINFO take double-precision floating point values.

LSgetStocParSample ()

Description:

Get a handle for the LSsample object associated with the specified stochastic parameter.

Returns:

A reference to an instance of LSsample object.

Prototype:

pLSsample	LSgetStocParSample (pLSmodel pModel, int iStv, int iRow, int jCol, int * nErrorCode)

Input Arguments:

Name	Description
pModel	A reference to an instance of LSmodel object.
iStv	An integer specifying the index of stochastic parameter in the instruction list. It should be ignored if (iRow,jCol) is specified.
iRow	An integer specifying the row index of the stochastic parameter. It should be ignored if `iStv` will be specified.
jCol	An integer specifying the column index of the stochastic parameter. It should be ignored if `iStv` will be specified.
nErrorCode	A reference to an integer error code.

Distribution Function Macros

Symbol	Value	Distribution Parameters		
		Param 1	**Param 2**	**Param 3**
Parametric Discrete Distributions				
		no. of trials	success prob.	N/A
LSDIST_TYPE_BINOMIAL	701	[0, +inf)	[0, 1]	N/A
		r- factor	success prob.	N/A
LSDIST_TYPE_NEGATIVE_BINOMIAL	704	(0, +inf)	(0, 1)	N/A
		success prob.	N/A	N/A
LSDIST_TYPE_GEOMETRIC	705	(0, 1]	N/A	N/A
		mean	N/A	N/A
LSDIST_TYPE_POISSON	706	(0, +inf)	N/A	N/A
		p-factor	N/A	N/A
LSDIST_TYPE_LOGARITHMIC	707	(0, 1)	N/A	N/A
		total pop. (N)	sample size (n)	defective factor (m)
LSDIST_TYPE_HYPER_GEOMETRIC	708	[0, +inf)	[0, N]	[0, N]
Parametric Continuous Distributions				
		Shape 1	Shape 2	N/A
LSDIST_TYPE_BETA	801	(0,+inf)	(0,+inf)	N/A
		location	scale	N/A
LSDIST_TYPE_CAUCHY	802	(-inf, +inf)	(0,+inf)	N/A
		deg. of freedom	N/A	N/A
LSDIST_TYPE_CHI_SQUARE	803	(0,+inf)	N/A	N/A
		Rate	N/A	N/A
LSDIST_TYPE_EXPONENTIAL	804	(0,+inf)	N/A	N/A
		deg. of freedom 1	deg. of freedom 2	N/A
LSDIST_TYPE_F_DISTRIBUTION	805	(0,+inf)	(0,+inf)	N/A
LSDIST_TYPE_GAMMA	806	shape	scale	N/A

		(0,+inf)	(0,+inf)	N/A
		location	scale	N/A
LSDIST_TYPE_GUMBEL	807	(-inf, +inf)	(0,+inf)	N/A
		location	scale	N/A
LSDIST_TYPE_LAPLACE	808	(-inf, +inf)	(0,+inf)	N/A
		location	scale	N/A
LSDIST_TYPE_LOGNORMAL	809	(-inf, +inf)	(0,+inf)	N/A
		location	scale	N/A
LSDIST_TYPE_LOGISTIC	810	(-inf, +inf)	(0,+inf)	N/A
		mean	standard deviation	N/A
LSDIST_TYPE_NORMAL	811	(-inf, +inf)	(0,+inf)	N/A
		scale	shape	N/A
LSDIST_TYPE_PARETO	812	(0,+inf)	(0,+inf)	N/A
		deg. of freedom	N/A	N/A
LSDIST_TYPE_STUDENTS_T	814	(0,+inf)	N/A	N/A
		lower limit (a)	upper limit (b)	mode (c)
LSDIST_TYPE_TRIANGULAR	815	(-inf, b]	[a, +inf)	[a, b]
		lower limit (a)	upper limit (b)	N/A
LSDIST_TYPE_UNIFORM	816	(-inf, b]	[a, +inf)	N/A
		shape	scale	N/A
LSDIST_TYPE_WEIBULL	817	(0,+inf)	(0,+inf)	N/A
Customized Distributions				
LSDIST_TYPE_DISCRETE	702	N/A	N/A	N/A
LSDIST_TYPE_DISCRETE_BLOCK	703	N/A	N/A	N/A
LSDIST_TYPE_LINTRAN_BLOCK	709	N/A	N/A	N/A
LSDIST_TYPE_SUB_BLOCK	710	N/A	N/A	N/A
LSDIST_TYPE_SUB	711	N/A	N/A	N/A
LSDIST_TYPE_USER	712	N/A	N/A	N/A

Chapter 3: Solving Linear Programs

In this chapter, we demonstrate the use of LINDO API to build and solve a very simple model. We will give examples written in both C and Visual Basic.

Recall the simple programming example from Chapter 1. It is a small product mix model that appears as follows:

```
Maximize:    20 * A + 30 * C
Subject to:
                    A           <=  60
                              C  <=  50
                    A +  2 * C  <= 120
```

The optimal objective value for this model is 2100, with $A = 60$ and $C = 30$.

Solving such a problem with LINDO API involves the following steps:

1. Create a LINDO environment.
2. Create a model in the environment.
3. Specify the model.
4. Perform the optimization.
5. Retrieve the status and model solution.
6. Delete the LINDO environment.

We illustrate each of these steps for both C and Visual Basic.

A Programming Example in C

In this section, we will illustrate the use of LINDO API to build and solve the small model discussed above. The code for this example is contained in the file \lindoapi\samples\c\samp1\samp1.c. The contents of this file are reproduced below:

```
/* samp1.c

  A C programming example of interfacing with the
  LINDO API.

  The problem:
     MAX = 20 * A + 30 * C
     S.T.       A +  2 * C  <= 120
                A           <=  60
                          C  <=  50

   Solving such a problem with the LINDO API involves
   the following steps:
```

```
      1. Create a LINDO environment.
      2. Create a model in the environment.
      3. Specify the model.
      4. Perform the optimization.
      5. Retrieve the status and model solution.
      6. Delete the LINDO environment.
*/
#include <stdlib.h>
#include <stdio.h>

/* LINDO API header file is located under \lindoapi\include */
#include "lindo.h"

/* Define a macro to declare variables for error checking */
#define APIERRORSETUP
   int nErrorCode;
   char cErrorMessage[LS_MAX_ERROR_MESSAGE_LENGTH]

/* Define a macro to do our error checking */
#define APIERRORCHECK
   if (nErrorCode)
   {
      if ( pEnv)
      {
         LSgetErrorMessage( pEnv, nErrorCode,
          cErrorMessage);
         printf("Errorcode=%d:  %s\n", nErrorCode,
          cErrorMessage);
      } else {
         printf( "Fatal Error\n");
      }
      exit(1);
   }

/* main entry point */
int main()
{
   APIERRORSETUP;
/* Number of constraints */
   int nM = 3;
/* Number of variables */
   int nN = 2;
/* declare an instance of the LINDO environment object */
   pLSenv pEnv;
/* declare an instance of the LINDO model object */
   pLSmodel pModel;

   int nSolStatus;
   char MY_LICENSE_KEY[1024];

/* >>> Step 1 <<< Create a model in the environment. */
   nErrorCode = LSloadLicenseString(
               "../../../license/lndapi60.lic",MY_LICENSE_KEY);
   pEnv = LScreateEnv ( &nErrorCode, MY_LICENSE_KEY);
   if ( nErrorCode == LSERR_NO_VALID_LICENSE)
```

```
   {
      printf( "Invalid License Key!\n");
      exit( 1);
   }
   APIERRORCHECK;

/* >>> Step 2 <<< Create a model in the environment. */
   pModel = LScreateModel ( pEnv, &nErrorCode);
   APIERRORCHECK;
   {
/* >>> Step 3 <<< Specify the model.

 To specify our model, we make a call to LSloadLPData,
  passing it:
 - A pointer to the model which we are specifying(pModel)
 - The number of constraints in the model
 - The number of variables in the model
 - The direction of the optimization (i.e. minimize or
 -  maximize)
 - The value of the constant term in the objective (may
    be zero)
 - The coefficients of the objective function
 - The right-hand sides of the constraints
 - The types of the constraints
 - The number of nonzeros in the constraint matrix
 - The indices of the first nonzero in each column
 - The length of each column
 - The nonzero coefficients
 - The row indices of the nonzero coefficients
 - Simple upper and lower bounds on the variables
*/

/* The direction of optimization */
      int nDir = LS_MAX;
/* The objective's constant term */
      double dObjConst = 0.;
/* The coefficients of the objective function */
      double adC[2] = { 20., 30.};
/* The right-hand sides of the constraints */
      double adB[3] = { 120., 60., 50.};
/* The constraint types */
      char acConTypes[3] = {'L', 'L', 'L'};
/* The number of nonzeros in the constraint matrix */
      int nNZ = 4;
/* The indices of the first nonzero in each column */
      int anBegCol[3] = { 0, 2, nNZ};
/* The length of each column.  Since we aren't leaving
    any blanks in our matrix, we can set this to NULL */
      int *pnLenCol = NULL;
/* The nonzero coefficients */
      double adA[4] = { 1., 1., 2., 1.};
/* The row indices of the nonzero coefficients */
      int anRowX[4] = { 0, 1, 0, 2};
/* Simple upper and lower bounds on the variables.
    By default, all variables have a lower bound of zero
    and an upper bound of infinity.  Therefore pass NULL
```

```
    pointers in order to use these default values. */
      double *pdLower = NULL, *pdUpper = NULL;
/* We have now assembled a full description of the model.
    We pass this information to LSloadLPData with the
    following call. */
      nErrorCode = LSloadLPData( pModel, nM, nN, nDir,
       dObjConst, adC, adB, acConTypes, nNZ, anBegCol,
       pnLenCol, adA, anRowX, pdLower, pdUpper);
      APIERRORCHECK;
  }

 /* >>> Step 4 <<< Perform the optimization */
   nErrorCode = LSoptimize( pModel,
    LS_METHOD_PSIMPLEX, &nSolStatus);
   APIERRORCHECK;

   if (nSolStatus == LS_STATUS_OPTIMAL ||
       nSolStatus == LS_STATUS_BASIC_OPTIMAL)
   {
 /* >>> Step 5 <<< Retrieve the solution */
      int i;
      double adX[ 2], dObj;
 /* Get the value of the objective */
      nErrorCode = LSgetInfo( pModel, LS_DINFO_POBJ, &dObj) ;
      APIERRORCHECK;
      printf( "Objective Value = %g\n", dObj);
 /* Get the variable values */
      nErrorCode = LSgetPrimalSolution ( pModel, adX);
      APIERRORCHECK;
      printf ("Primal values \n");
      for (i = 0; i < nN; i++) printf( " x[%d] = %g\n", i,adX[i]);
      printf ("\n");
   }
   else
   {
     /* see include\lindo.h for status definitions */
     printf( "Optimal solution was not"
       " found -- status: %d\n", nSolStatus);
   }

 /* >>> Step 6 <<< Delete the LINDO environment */
   nErrorCode = LSdeleteModel( &pModel);
   nErrorCode = LSdeleteEnv( &pEnv);

  /* Wait until user presses the Enter key */
   printf("Press <Enter> ...");
   getchar();
}
```

\lindoapi\samples\c\samp1\samp1.c

The C header file *lindo.h* must be included in each C source file that makes any calls to LINDO API. This file contains definitions of all LINDO data structures, macros, and function prototypes. This is done in our sample with the following code:

```
/* LINDO API header file */
#include "lindo.h"
```

Next, the license key is read into a local string using the following code fragment.

```
nErrorCode = LSloadLicenseString(
             "../../../license/lndapi60.lic",MY_LICENSE_KEY);
```

The sample code then defines the macros *APIERRORSETUP* and *APIERRORCHECK* that are used to streamline error checking after calls to LINDO API. If an error is encountered after a call to a LINDO API routine, the *APIERRORCHECK* macro will cause the application to immediately cease execution.

As mentioned above, the first two major steps in a typical application calling LINDO API are: 1) creating a LINDO environment object, and 2) creating a model object within the environment. The following code segment does this:

```
/* declare an instance of the LINDO environment object */
   pLSenv pEnv;

/* declare an instance of the LINDO model object */
   pLSmodel pModel;

   int nSolStatus;
   char MY_LICENSE_KEY[1024];

/* >>> Step 1 <<< Create a model in the environment */
nErrorCode = LSloadLicenseString(
            "../../../license/lndapi60.lic",MY_LICENSE_KEY);
   pEnv = LScreateEnv ( &nErrorCode, MY_LICENSE_KEY);
   if ( nErrorCode == LSERR_NO_VALID_LICENSE)
   {
      printf( "Invalid License Key!\n");
      exit( 1);
   }
   APIERRORCHECK;

/* >>> Step 2 <<< Create a model in the environment. */
   pModel = LScreateModel ( pEnv, &nErrorCode);
   APIERRORCHECK;
```

The environment data type, *pLSenv*, and the model data type, *pLSmodel*, are both defined in the *lindo.h* header file. A call to *LScreateEnv*() creates the LINDO environment. The second argument to *LScreateEnv*() is the local sting variable *MY_LICENSE_KEY* that holds the license key read from *lndapi60.lic* file. Immediately after the call to *LScreateEnv*(), a specific error check is done to trap the condition of an invalid license key. Finally, the model object is created with a call to *LScreateModel*().

The next step is to define the model. This is generally the most involved of the steps. The model definition code in this example is as follows:

```
/* The direction of optimization */
      int nDir = LS_MAX;

/* The objective's constant term */
      double dObjConst = 0.;

/* The coefficients of the objective function */
      double adC[2] = { 20., 30.};

/* The right-hand sides of the constraints */
      double adB[3] = { 60., 50., 120.};

/* The constraint types */
      char acConTypes[3] = {'L', 'L', 'L'};
```

```
/* The number of nonzeros in the constraint matrix */
      int nNZ = 4;

/* The indices of the first nonzero in each column */
      int anBegCol[3] = { 0, 2, nNZ};

/* The length of each column. Since we aren't leaving
    any blanks in our matrix, we can set this to NULL */
      int *pnLenCol = NULL;

/* The nonzero coefficients */
      double adA[4] = { 1., 1., 1., 2.};

/* The row indices of the nonzero coefficients */
      int anRowX[4] = { 0, 2, 1, 2};

/* Simple upper and lower bounds on the variables.
    By default, all variables have a lower bound of zero
    and an upper bound of infinity. Therefore pass NULL
    pointers in order to use these default values. */
      double *pdLower = NULL, *pdUpper = NULL;

/* We have now assembled a full description of the model.
    We pass this information to LSloadLPData with the
    following call. */
      nErrorCode = LSloadLPData( pModel, nM, nN, nDir,
       dObjConst, adC, adB, acConTypes, nNZ, anBegCol,
       pnLenCol, adA, anRowX, pdLower, pdUpper);
      APIERRORCHECK;
```

First, the direction of the objective is set with the following:

```
/* The direction of optimization */
      int nDir = LS_MAX;
```

Had the problem been a minimization type, *LS_MIN* would have been used instead.

This model does not have a constant term in the objective, so it is set to zero:

```
/* The objective's constant term */
      double dObjConst = 0.;
```

The model's objective coefficients are placed into an array:

```
/* The coefficients of the objective function */
      double adC[2] = { 20., 30.};
```

The constraint right-hand side values are placed into an array:

```
/* The right-hand sides of the constraints */
      double adB[3] = { 60., 50., 120.};
```

The constraint types are placed into an array:

```
/* The constraint types */
      char acConTypes[3] = {'L', 'L', 'L'};
```

The three constraints in this model are less-than-or-equal-to constraints. Thus, all the constraint type codes are set to be "L". Had any of the constraints been greater-than-or-equal-to, equality, or neutral, the constraint type code would have been set to "G", "E", or "N", respectively.

The number of nonzero coefficients in the constraint matrix is stored:

```
/* The number of nonzeros in the constraint matrix */
      int nNZ = 4;
```

The index of the first nonzero element in each column is placed into an array:

```
/* The indices of the first nonzero in each column */
      int anBegCol[3] = { 0, 2, nNZ};
```

Note that zero based indices are being used. This array index must have one more element than the number of variables. The extra element must point to where any new column would start in the nonzero coefficient matrix.

The next step, is to perform the optimization of the model. This is accomplished with the following call to *LSoptimize*():

```
/* >>> Step 4 <<< Perform the optimization */
   nErrorCode = LSoptimize( pModel,
    LS_METHOD_PSIMPLEX, & nSolStatus);
   APIERRORCHECK;
```

LSoptimize() takes three arguments. The first is the pointer to the model object you wish to optimize. The second is the index of the type of solver you wish to use. In this case, the primal simplex solver was selected by setting the second argument to `LS_METHOD_PSIMPLEX`. Alternative types of solvers available for linear models include dual simplex and barrier (if licensed). The third argument is a pointer to return the status of the solution.

Once the model is solved, the next step is to retrieve the components of the solution that are of interest to your particular application. In this example, the objective value and the variable values are displayed. First, check whether *LSoptimize*() successfully computed an optimal solution by examining the value of the status variable *nSolStatus*. Provided that an optimal solution is available, a call to *LSgetInfo*() with macro *LS_DINFO_POBJ* fetches the (primal) objective value, while a call to *LSgetPrimalSolution*() retrieves the variable values:

```
if (nSolStatus == LS_STATUS_OPTIMAL ||
    nSolStatus == LS_STATUS_BASIC_OPTIMAL)
{
/* >>> Step 5 <<< Retrieve the solution */
      int i;
      double adX[ 2], dObj;

/* Get the value of the objective */
      nErrorCode = LSgetInfo( pModel, LS_DINFO_POBJ, &dObj) ;
      APIERRORCHECK;

      printf( "Objective Value = %g\n", dObj);

/* Get the variable values */
      nErrorCode = LSgetPrimalSolution ( pModel, adX);
      APIERRORCHECK;

      printf ("Primal values \n");
      for (i = 0; i < nN; i++) printf( " x[%d] = %g\n", i, adX[i]);
      printf ("\n");
}
```

As our last step, the LINDO environment is deleted with a call to *LSdeleteEnv*():

```
/* >>> Step 6 <<< Delete the LINDO environment */
      nErrorCode = LSdeleteEnv( &pEnv);
```

This allows LINDO to free up all data structures allocated to the environment and all of the environment's associated models.

The next section goes through the steps required for compiling and linking this program using Visual C++ 6.0. However, keep in mind that any C development environment should be able to successfully link the code above with LINDO API.

This application will be built using the *nmake* utility supplied with Visual C++. The mechanics for performing the build are illustrated in the DOS command line session below, where user input is displayed in bold type:

```
C:\>cd \lindoapi\samples\c\samp1

C:\lindoapi\samples\c\samp1>dir

 Volume in drive C has no label.
 Volume Serial Number is 1833-D1E6

 Directory of C:\lindoapi\samples\c\samp1

11/25/02  12:00p        <DIR>          .
11/25/02  12:00p        <DIR>          ..
11/25/02  12:00p                 1,347 makefile.unx
11/25/02  12:00p                 1,371 makefile.win
11/25/02  12:00p                 5,307 samp1.c


11/25/02  12:00p                 4,285 samp1.dsp
11/25/02  12:00p                   533 samp1.dsw
11/25/02  12:00p                36,864 samp1.exe

               8 File(s)         48,923 bytes

                           5,553,143,808 bytes free
C:\lindoapi\samples\c\samp1>del samp1.exe

C:\lindoapi\samples\c\samp1>command /e:32000

Microsoft(R) Windows 98
   (C)Copyright Microsoft Corp 1981-1998.

C:\lindoapi\samples\c\samp1>vcvars32
Setting environment for using Microsoft Visual C++ tools.

C:\lindoapi\samples\c\samp1>nmake -f makefile.win

Microsoft (R) Program Maintenance Utility   Version 6.00.8168.0
Copyright (C) Microsoft Corp 1988-1998. All rights reserved.

        cl -c -D_LINDO_DLL_ -I"..\..\..\include" -I"..\..\..\license"
samp1.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for
80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.

samp1.c
        cl samp1.obj ..\..\..\lib\win32\lindo6_0.lib -Fesamp1.exe
```

```
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for
80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.

Microsoft (R) Incremental Linker Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/out:samp1.exe
samp1.obj
..\..\..\lib\win32\lindo6_0.lib

C:\lindoapi\samples\c\samp1>samp1
Objective Value = 2100
Primal values
 x[0] = 60
 x[1] = 30

Press <Enter> ...
```

The following seven commands were issued to build and run the application:

- **cd \lindoapi\samples\c\samp1**– This selects the directory where the sample code is stored. This assumes that you placed LINDO API into the default subdirectory titled "lindoapi".
- **dir** – A directory listing is requested from DOS. The two key files are *samp1.c* (the source file) and *makefile.win* (the input file for the *nmake* utility). The *samp1.exe* file is a copy of the executable that was supplied with LINDO API.
- **del samp1.exe** – Since the file will be built, the old copy is removed.
- **command /e:32000** – This loads a new copy of the command line processor. This is done in order to increase the space allocated to the environment with the /e:32000 switch. This allocates enough space in the environment to store all the environment variables required by the Visual C++ compiler.
- **vcvars32** – This runs the *vcvars32.bat* batch job supplied by Visual C++ that configures the environment for use of the command line versions of the compiler tools. If this command fails, it is probably due to not having *vcvars32.bat* on your search path. Search your hard drive for *vcvars32.bat*, and then move it to either a directory on the search path or to the directory where this sample is located. If you are unable to find *vcvars32.bat*, you will need to reinstall Visual C++.
- **nmake –f makefile.win** – This command invokes the make utility that uses the input supplied in *makefile.win* to perform the compiling and linking of our sample application. The details of the steps contained in this file are discussed below.
- **samp1** – Here the actual sample application is run. As predicted, the optimal objective value is 2100, variable 1 has a value of 60, and variable 2 has a value of 30.

The contents of the make utility input file, *makefile.win*, are listed below. Users on Unix-like platforms should refer to *makefile.unx*:

```
EXAMPLE= samp1
IFLAGS = -I"..\..\..\include" -I"..\..\..\license"
DFLAGS = -D_LINDO_DLL_

all : $(EXAMPLE).obj $(EXAMPLE).exe
$(EXAMPLE).obj : $(EXAMPLE).c
       cl -c $(DFLAGS) $(IFLAGS) $(EXAMPLE).c

$(EXAMPLE).exe : ..\..\..\lib\win32\lindo6_0.lib $(EXAMPLE).obj
```

```
        cl $(EXAMPLE).obj ..\..\..\lib\win32\lindo6_0.lib -
Fe$(EXAMPLE).exe
```

The first and second lines designate the name of the executable and the paths to include directories. The third line defines the preprocessor macro *_LINDO_DLL_*. This definition modifies the behavior of the *lindo.h* header file in order to allow access to LINDO API as a DLL. Users on platforms other than Windows should omit this definition.

The fourth line uses the "all" pseudo target to specify the build order. The following set of instructions listing *samp1.obj* as the target invokes the command line compiler to compile our source file. The next directive listing *samp1.exe* as the target links the object code with the LINDO API import library to build the completed application.

If you would prefer to build this application using the Visual C++ 6.0 IDE, you should follow these steps:

1. Start Visual C++ 6.0.
2. Issue the *File|New* command.
3. Do the following in the "New" dialog box: select the "Project" tab, click on "Win32 Console Application", in the "Project Name" edit field enter "MySample", in the "Location" edit field enter *\lindoapi\samples\c\samp1*, and, finally, click the *OK* button.
4. Click the *Finish* button in the "Win32 Console Application" dialog.
5. Click the *OK* button to clear the "New Project Information" dialog.
6. Run the *Project|Add to Project|Files* command and add *\lindoapi\samples\c\samp1\samp1.c* to the project.
7. Run the *Project|Add to Project|Files* command and add *\lindoapi\lib\win32\lindo6_0.lib* to the project.
8. Run the *Project|Settings* command, select "All Configurations" from the "Settings For" drop down list box, select the C/C++ tab, select "General" from the "Category" list box, and in the "Preprocessor definitions" box add *_LINDO_DLL_* to the list of definitions. Click the *OK* button.
9. Once again, run the *Project|Settings* command, select "All Configurations" from the "Settings For" drop down list box, select the C/C++ tab, select "Preprocessor" from the "Category" list box, and in the "Additional include directories" box add "\lindoapi\include" and "\lindoapi\license" (without quotes and separated by a comma). Click the *OK* button.
10. Run the *File|Save Workspace* command.
11. Run the *Build|Rebuild All* command to build the executable.
12. Run the *Build|Start Debug|Go* command to run the sample application.

A Programming Example in Visual Basic

The overall design and code of a program in Visual Basic is quite similar to the C example. Analogous to the inclusion of *lindo.h* in our C example, the Visual Basic project includes a module titled *lindo.bas*, which facilitates access to LINDO API. A copy of *lindo.bas* may be found in the main LINDO API folder. Add *lindo.bas* to projects with the *Project|Add Module* command in VB.

There are differences in syntax between the C and VB code. Calls to LINDO API within Visual Basic are made using Visual Basic type variables. These types differ from the types specified by LINDO API C-language function prototypes detailed in Chapter 2, *Function Definitions*. The following chart of conversions shows how to invoke C routines by passing the appropriate arguments in calls.

If a LINDO routine expects…	**Then from VB pass…**
An int	A Long
A double	A Double
A pointer to a numeric value	Pass the variable as you normally would
A numeric array	Pass the first element of the array
A null pointer to a numeric value	Pass 'ByVal 0'
A character array	Pass a String
A null pointer to a character array	Pass the constant *vbNullString*

The difference in types between C and VB affects use of several routines returning pointers to a LINDO environment or model object. For example, *LScreateEnvironment*() and *LScreateModel*() return pointers to environment and model objects, respectively. Neither of these objects can be defined in Visual Basic (because they contain pointers). Fortunately, the user of LINDO API never has to directly access or modify these objects. All we need is a pointer to them, which can be conveyed in Visual Basic code with a Long variable. Wherever a pointer to an environment or a model is needed, a Long variable can be substituted in its place.

Using VB, the product mix model listed at the beginning of this chapter will be solved once again. The VB 6.0 project for this example may be found in *\lindoapi\samples\vb\samp1\samplevb.vbp*, and may be loaded into VB 6.0 with the *File|Open Project* command. The code for solving the model is listed below:

```
'  A VB programming example of interfacing with the
'  LINDO API.
'
'  the problem:
'     Max = 20 * A + 30 * C
'     S.T.       A +  2 * C  <= 120
'                A           <=  60
'                          C  <=  50
'   Solving such a problem with the LINDO API involves
'   the following steps:
'      1. Create a LINDO environment.
'      2. Create a model in the environment.
'      3. Specify the model.
'      4. Perform the optimization.
'      5. Retrieve the solution.
```

```
'      6. Delete the LINDO environment.

Option Explicit

Private Sub Command1_Click()
  'Declarations
  Dim con_type As String
  Dim env As Long
  Dim errorcode As Long
  Dim i As Long
  Dim m As Long
  Dim n As Long
  Dim nz As Long
  Dim prob As Long
  Dim Abegcol() As Long
  Dim Arowndx() As Long
  Dim Acoef() As Double
  Dim b() As Double
  Dim c() As Double
  Dim obj As Double
  Dim x() As Double
  Dim LicenseKey As String * LS_MAX_ERROR_MESSAGE_LENGTH
' Name data
  Dim szTitle, szObjName, szRhsName, szRngName, szBndname As String
  Dim szConNames() As String
  Dim szVarNames() As String
' Auxiliary byte arrays for keeping variable and constraint name
' data for keeping
  Dim acConNames() As Byte
  Dim acVarNames() As Byte
' Pointer arrays for storing the address of each name within the byte
' arrays. These pointers will be passed to LINDO API
  Dim pszConNames() As Long
  Dim pszVarNames() As Long

'>>> Step 1 <<<:  Create a LINDO environment.
  errorcode = LSloadLicenseString("\lindoapi\license\lndapi60.lic",
              LicenseKey)
  Call CheckErr(env, errorcode)
  env = LScreateEnv(errorcode, LicenseKey)
  If (errorcode > 0) Then
     MsgBox ("Unable to create environment.")
     End
  End If

'>>> Step 2 <<<:  Create a model in the environment.
  prob = LScreateModel(env, errorcode)
  Call CheckErr(env, errorcode)

'>>> Step 3 <<<:  Specify the model.
  'Set the problem sizes
  'number of constraints
  m = 3
  'number of variables
  n = 2
  'objective coefficients
```

```
ReDim c(n)
c(0) = 20
c(1) = 30
'right-hand-sides of constraints
ReDim b(m)
b(0) = 120
b(1) = 60
b(2) = 50
'constraint types
con_type = "LLL"
'index of first nonzero in each column
ReDim Abegcol(n + 1)
Abegcol(0) = 0
Abegcol(1) = 2
Abegcol(2) = 4
'number of nonzeros in constraint matrix
nz = 4
'the nonzero coefficients
ReDim Acoef(nz)
Acoef(0) = 1
Acoef(1) = 1
Acoef(2) = 2
Acoef(3) = 1
'the row indices of the nonzeros
ReDim Arowndx(nz)
Arowndx(0) = 0
Arowndx(1) = 1
Arowndx(2) = 0
Arowndx(3) = 2
' Load LP data
errorcode = LSloadLPData(prob, m, n, LS_MAX, 0, _
 c(0), b(0), con_type, nz, Abegcol(0), ByVal 0, _
 Acoef(0), Arowndx(0), ByVal 0, ByVal 0)
Call CheckErr(env, errorcode)
' name data
szTitle = "SAMP1"
szObjName = "OBJ"
szRhsName = "RHS"
szRngName = "RNG"
szBndname = "BND"
' local arrays for variable and constraint names
ReDim szConNames(m)
ReDim szVarNames(n)
Dim szConNamesLen As Long, szVarNamesLen As Long
szConNames(0) = "Cons0"
szConNames(1) = "Cons1"
szConNames(2) = "Cons2"
For i = 0 To m - 1
  szConNamesLen = szConNamesLen + Len(szConNames(i)) + 1
Next
szVarNames(0) = "VarA"
szVarNames(1) = "VarC"
For i = 0 To n - 1
  szVarNamesLen = szVarNamesLen + Len(szVarNames(i)) + 1
Next
' byte arrays to keep name data
```

```
    ReDim acConNames(szConNamesLen)
    ReDim acVarNames(szVarNamesLen)
    ' pointer arrays for keeping addresses of each name
    ' located in the byte arrays
    ReDim pszConNames(m)
    ReDim pszVarNames(n)
    ' parse string arrays to byte arrays and record pointers (source:
    ' Strutil.bas)
    Call NameToPtr(acConNames, pszConNames, szConNames, m)
    Call NameToPtr(acVarNames, pszVarNames, szVarNames, n)
    ' pass names
    errorcode = LSloadNameData(prob, szTitle, szObjName, szRhsName,
                szRngName, szBndname, _
    pszConNames(0), pszVarNames(0))
    Call CheckErr(env, errorcode)
    ' Export the model in LINDO File format
    Dim LindoFile As String
    LindoFile = "samp1.mps"
    Call LSwriteMPSFile(prob, LindoFile, LS_FORMATTED_MPS)

    '>>> Step 4 <<<:  Perform the optimization.
    errorcode = LSoptimize(prob, LS_METHOD_PSIMPLEX, ByVal 0)
    Call CheckErr(env, errorcode)

    '>>> Step 5 <<<:  Retrieve the solution.
    'Print the objective value and primals
    errorcode = LSgetInfo(prob, LS_DINFO_POBJ, obj)
    Call CheckErr(env, errorcode)
    ReDim x(n)
    errorcode = LSgetPrimalSolution(prob, x(0))
    Call CheckErr(env, errorcode)
    MsgBox ("Objective value: " & obj & vbCrLf & _
     "Primal values: A=" & x(0) & ", C=" & x(1))
    errorcode = LSsetModelIntParameter(prob,
  LS_IPARAM_SOL_REPORT_STYLE, 0)
    errorcode = LSwriteSolution(prob, "samp1.sol")
    Call LSdeleteModel(prob)

    '>>> Step 6 <<< Delete the LINDO environment.
    Call LSdeleteEnv(env)
  End Sub

  Public Sub CheckErr(env As Long, errorcode As Long)
  ' Checks for an error condition.  If one exists, the
  '  error message is displayed then the application
  '  terminates.
     If (errorcode > 0) Then
        Dim message As String
        message = String(LS_MAX_ERROR_MESSAGE_LENGTH, _
         vbNullChar)
        Call LSgetErrorMessage(env, errorcode, message)
        MsgBox (message)
        End
     End If
  End Sub
```

```
Private Sub Form_Load()
Dim szVernum As String * LS_MAX_ERROR_MESSAGE_LENGTH
Dim szBuildDate As String * LS_MAX_ERROR_MESSAGE_LENGTH
Call LSgetVersionInfo(szVernum, szBuildDate)

Label2.Caption = "LINDO API  Version  " & szVernum
Label1.Caption = "Max = 20 A + 30 C " & vbNewLine & vbNewLine & _
                 "S.T.     A +  2 C  <= 120 " & vbNewLine & _
                 "          A    <=  60 " & vbNewLine & _
                 "             C <=  50 " & vbNewLine & vbNewLine & _
                 "  A , C are nonnegative    "
End Sub
```

\lindoapi\samples\vb\samp1\samplevb.frm

As mentioned above, the first two major steps in a typical application calling LINDO API are: 1) creating a LINDO environment object, and 2) creating a model object within the environment. This is done with the following code segment:

```
''>>> Step 1 <<<:  Create a LINDO environment.
  errorcode = LSloadLicenseString("\lindoapi\license\lndapi60.lic",
LicenseKey)
  Call CheckErr(env, errorcode)
  env = LScreateEnv(errorcode, LicenseKey)
  If (errorcode > 0) Then
     MsgBox ("Unable to create environment.")
     End
  End If

'>>> Step 2 <<<:  Create a model in the environment.
  prob = LScreateModel(env, errorcode)
  Call CheckErr(env, errorcode)
```

The next step is to call *LScreateModel*() to create a model object in the newly created environment. After the call to *LScreateModel*(), a routine called *CheckErr*() is called. This routine is defined at the bottom of our code module. The code for *CheckErr*() has been reproduced below:

```
Public Sub CheckErr(env As Long, errorcode As Long)

' Checks for an error condition.  If one exists, the
'  error message is displayed then the application
'  terminates.

   If (errorcode > 0) Then
      Dim message As String
      message = String(LS_MAX_ERROR_MESSAGE_LENGTH, _
       vbNullChar)
      Call LSgetErrorMessage(env, errorcode, message)
      MsgBox (message)
      End
   End If

End Sub
```

CheckErr() is merely used to determine if LINDO API returned an error. If an error is returned, *CheckErr*() calls the API routine *LSgetErrorMessage*() to translate the error code into a text message. The message is displayed, and *CheckErr*() terminates the application.

The next step is to define the model. The model definition code is listed here:

```
'>>> Step 3 <<<:  Specify the model.
  'Set the problem sizes
  'number of constraints
  m = 3
  'number of variables
  n = 2
  'objective coefficients
  ReDim c(n)
  c(0) = 20
  c(1) = 30
  'right-hand-sides of constraints
  ReDim b(m)
  b(0) = 120
  b(1) = 60
  b(2) = 50
  'constraint types
  con_type = "LLL"
  'index of first nonzero in each column
  ReDim Abegcol(n + 1)
  Abegcol(0) = 0
  Abegcol(1) = 2
  Abegcol(2) = 4
  'number of nonzeros in constraint matrix
  nz = 4
  'the nonzero coefficients
  ReDim Acoef(nz)
  Acoef(0) = 1
  Acoef(1) = 1
  Acoef(2) = 2
  Acoef(3) = 1
  'the row indices of the nonzeros
  ReDim Arowndx(nz)
  Arowndx(0) = 0
  Arowndx(1) = 1
  Arowndx(2) = 0
  Arowndx(3) = 2
  ' Load LP data
  errorcode = LSloadLPData(prob, m, n, LS_MAX, 0, _
   c(0), b(0), con_type, nz, Abegcol(0), ByVal 0, _
   Acoef(0), Arowndx(0), ByVal 0, ByVal 0)
  Call CheckErr(env, errorcode)
  ' name data
  szTitle = "SAMP1"
  szObjName = "OBJ"
  szRhsName = "RHS"
  szRngName = "RNG"
  szBndname = "BND"
  ' local arrays for variable and constraint names
  ReDim szConNames(m)
  ReDim szVarNames(n)
  Dim szConNamesLen As Long, szVarNamesLen As Long
  szConNames(0) = "Cons0"
  szConNames(1) = "Cons1"
  szConNames(2) = "Cons2"
```

```
  For i = 0 To m - 1
    szConNamesLen = szConNamesLen + Len(szConNames(i)) + 1
  Next
  szVarNames(0) = "VarA"
  szVarNames(1) = "VarC"
  For i = 0 To n - 1
    szVarNamesLen = szVarNamesLen + Len(szVarNames(i)) + 1
  Next
  ' byte arrays to keep name data
  ReDim acConNames(szConNamesLen)
  ReDim acVarNames(szVarNamesLen)
  ' pointer arrays for keeping addresses of each name
  ' located in the byte arrays
  ReDim pszConNames(m)
  ReDim pszVarNames(n)
  ' parse string arrays to byte arrays and record pointers (source:
 ' Strutil.bas)
  Call NameToPtr(acConNames, pszConNames, szConNames, m)
  Call NameToPtr(acVarNames, pszVarNames, szVarNames, n)
  ' pass names
  errorcode = LSloadNameData(prob, szTitle, szObjName, szRhsName,
szRngName, szBndname, _
  pszConNames(0), pszVarNames(0))
  Call CheckErr(env, errorcode)
  ' Export the model in LINDO File format
  Dim LindoFile As String
  LindoFile = "samp1.mps"
  Call LSwriteMPSFile(prob, LindoFile, LS_FORMATTED_MPS)
```

First, the model's dimensions are stored:

```
'Set the problem sizes
 'number of constraints
 m = 3
 'number of variables
 n = 2
```

Then, the arrays are filled with the objective and right-hand side coefficients:

```
'objective coefficients
 ReDim c(n)
 c(0) = 20
 c(1) = 30
 'right-hand sides of constraints
 ReDim b(m)
 b(0) = 120
 b(1) = 60
 b(2) = 50
```

There are three constraints in the model, and all are of type less-than-or-equal-to. Thus, a string of three L's is stored to indicate this to the solver:

```
'constraint types
 con_type = "LLL"
```

Index of first nonzero in each column are stored next:

```
'index of first nonzero in each column
 ReDim Abegcol(n + 1)
 Abegcol(0) = 0
 Abegcol(1) = 2
 Abegcol(2) = 4
```

The constraint nonzero coefficients are stored next:

```
'number of nonzeros in constraint matrix
 nz = 4
'the nonzero coefficients
 ReDim Acoef(nz)
 Acoef(0) = 1
 Acoef(1) = 1
 Acoef(2) = 2
 Acoef(3) = 1
```

There are four nonzeros in the constraints—two for column *A* and two for column *C*. Note that the nonzero coefficients for column *A* (1,1) are passed first. The nonzeros for column *C* (2,1) are passed next.

Next, the row indices for the constraint nonzeros are stored:

```
'the row indices of the nonzeros
 ReDim Arowndx(nz)
 Arowndx(0) = 0
 Arowndx(1) = 1
 Arowndx(2) = 0
 Arowndx(3) = 2
```

Note that the indices are zero-based, so a nonzero in the first constraint has a row index of 0.

Finally, all the data is passed off to LINDO API with the following call to *LSloadLPData*():

```
errorcode = LSloadLPData(prob, m, n, LS_MAX, 0, _
 c(0), b(0), con_type, nz, Abegcol(0), ByVal 0, _
 Acoef(0), Arowndx(0), ByVal 0, ByVal 0)
Call CheckErr(env, errorcode)
```

Note that the fourth argument has been explicitly set to be *LS_MAX* to indicate that the objective is to be maximized. Another interesting aspect of this call is that arguments 11, 14, and 15 have been set to "ByVal 0". These arguments respectively correspond to the column-nonzero-count array, variable-lower-bound array, and variable-upper-bound array. A column-nonzero-count array is not needed, because our nonzeros have been stored in a dense manner. Therefore, the column-nonzero count is inferred from the other data. The default bounds for variables are zero to infinity, which are appropriate for this example. Thus, the two-variable bound arguments are also superfluous. By setting these arguments to "ByVal 0", a C-style null pointer is mimicked. This indicates that values are not supplied.

Now that the model has been defined, the next step is to invoke the solver. This is done with the following call to *LSoptimize*():

```
 '>>> Step 4 <<<:  Perform the optimization.
 errorcode = LSoptimize(prob, LS_METHOD_PSIMPLEX,ByVal 0)
 Call CheckErr(env, errorcode)
```

As our next step, the solution from LINDO is retrieved:

```
'>>> Step 5 <<<:  Retrieve the solution.
  'Print the objective value and primals
errorcode = LSgetInfo(prob, LS_DINFO_POBJ, obj)
Call CheckErr(env, errorcode)
ReDim x(n)
errorcode = LSgetPrimalSolution(prob, x(0))
Call CheckErr(env, errorcode)
MsgBox ("Objective value: " & obj & vbCrLf & _
 "Primal values: A=" & x(0) & ", C=" & x(1))
errorcode = LSsetModelIntParameter(prob,
            LS_IPARAM_SOL_REPORT_STYLE, 0)
errorcode = LSwriteSolution(prob, "samp1.sol")
Call LSdeleteModel(prob)
```

The objective is fetched with a call to *LSgetObjective*() and the variable values by calling *LSgetPrimalSolution*(). When this application is run, these values are posted in a dialog box as shown below:

The final step is to free up the memory allocated by LINDO API with a call to *LSdeleteEnv*():

```
'>>> Step 6 <<< Delete the LINDO environment.
Call LSDeleteEnv( env)
```

VB and Delphi Specific Issues:

Some of LINDO API's functions accept C-type NULL as a valid argument. Passing a NULL value would allow the associated argument to be left out of the scope of the action requested. For instance, consider the following use of LSgetBasis function using the C language.

```
{ // init
  int *panCstatus = malloc(nVars*sizeof(int));
  int *panRstatus = malloc(nCons*sizeof(int));
  int nErr = LSERR_NO_ERROR;
     ..
  // FIRST call to LSgetBasis
  nErr = LSgetBasis(pModel, panCstatus,  NULL);

  // SECOND call to LSgetBasis
  nErr = LSgetBasis(pModel, NULL, panRstatus );

  ..
  // clean
  free (panCstatus);
  free (panRstatus)
}
```

The first call to LSgetBasis retrieves the basis status of primal variables and places them in panCstatus vector. Here, the retrieval of basis status of the constraint slacks are skipped since a NULL value was passed as the third argument. In the second call, the basis status of primal variables was ignored in a similar fashion and the basis status of the constraint slacks were retrieved. This calling convention is common with most query and loading routines in LINDO API.

In certain programming languages (e.g. VB or Delphi), where NULL is not supported, this functionality can be achieved by following the steps below:

Step 1) Locate the function declaration under consideration in the header file associated with the language you are using (e.g. lindo.bas (VB), lindo.pas (Delphi)).
Step 2) Locate all the arguments that you want to pass a NULL value for.
Step 3) Modify the type of these arguments from 'by-reference' to 'by-value', whatever that would mean in terms of the underlying language.
Step 4) Go back to your application and pass a zero value for these arguments.

For instance, Step 3 would lead to the following definition of LSgetBasis() in lindo.bas (VB).

```
 Public Declare Function LSgetBasis _
Lib "LINDO6_0.DLL" (ByVal nModel As Long, _
                    ByRef anCstatus As Any, _
                    ByRef anRstatus As Any) As Long
```

A hypothetical VB application could then make the following calls

```
Redim anCstatus(nVars)
Redim anRstatus(nVars)
..
LSgetBasis(pModel, anCstatus, ByVal 0)
LSgetBasis(pModel, ByVal 0, anRstatus)
..
```

Similarly, the following modification to LSgetBasis() in lindo.pas would allow the same effect for Delphi.

```
function  LSgetBasis ( nModel    :  Integer;
                       anCstatus :  Integer;
                   Var anRstatus :  Integer) : Integer; stdcall;
external 'lindo6_0.dll';
```

The situation is handled in a similar fashion for string arrays, but with a little extra work. LINDO API functions that take string arrays as arguments require that all string arrays are converted to a C-type character array before they are passed. A simple utility module for VB performing this conversion is available as “lindoapi/include/strutil.bas”. Please refer to the sample VB application under “lindoapi/samples/vb/ samp1” for an illustration of how this interface is used to pass string arrays (or a NULL when needed) to the solver.

Chapter 4: Solving Mixed-Integer Programs

This chapter walks through an example of a *mixed-integer programming* (MIP) model. A MIP model restricts one or more variables to integer values.

MIP models require calling a few different routines from what would normally be called when solving a linear program (LP). This distinction provides the ability to maintain both the MIP solution and the continuous solution to a problem. The table below summarizes these differences:

Task	LP Routine	MIP Routine(s)
Loading formulation	LSloadLPData()	LSloadLPData() LSloadVarType()
Establish callback routine	LSsetCallback()	LSsetCallback() LSsetMIPCallback()
Solve	LSoptimize()	LSsolveMIP()
Get information in callback	LSgetCallbackInfo()	LSgetCallbackInfo() LSgetMIPCallbackInfo()
Get objective value	LSgetInfo()	LSgetInfo()
Get primals	LSgetPrimalSolution()	LSgetMIPPrimalSolution ()
Get slacks	LSgetSlacks()	LSgetMIPSlacks()
Get duals	LSgetDualSolution()	LSgetMIPDualSolution()
Get reduced costs	LSgetReducedCosts()	LSgetMIPReducedCosts()

As the table shows, loading a MIP formulation requires calling *LSloadVarType*() in addition to *LSloadLPData*(). The additional call to *LSloadVarType*() is made to identify the integer variables. An additional callback routine may be established for MIP models by calling *LSsetMIPCallback*(). The solver calls this additional callback routine every time a new integer solution is found. When retrieving information in callbacks, you may also be interested in *LSgetMIPCallbackInfo*(). This routine returns solver status information that is specific to MIP models.

This sample model is a staffing model that computes the optimal staffing levels for a small business. Specifically, it determines the number of employees to start on each day of the week to minimize total staffing costs, while meeting all staffing demands throughout the week. Since a fractional number of employees cannot start, the variables representing the number of employees are required to be integer.

Suppose you run the popular Pluto Dogs hot dog stand that is open seven days a week. Employees are hired to work a five-day workweek with two consecutive days off. Each employee receives the same weekly salary. You would like to develop an interactive application that would allow you to enter your staffing requirements for a week and then use this data to come up with a minimal cost staff schedule minimizing the total number of required employees, while still meeting (or exceeding) staffing requirements.

The model generated to solve this problem will have seven variables and seven constraints. The *i*-th variable represents the number of employees to start on the *i*-th day of the week. The *i*-th constraint will sum up the number of employees working on the *i*-th day of the week, and set this sum to be greater-than-or-equal-to the staff required on the *i*-th day of the week. The objective will simply minimize the sum of all employees starting throughout the week. The formulation for this model appears below:

```
MIN  M + T + W + R + F + S + N
SUBJECT TO
     M +           R + F + S + N >=
     M + T +           F + S + N >=
     M + T + W +           S + N >=
     M + T + W + R +           N >=
     M + T + W + R + F           >=
         T + W + R + F + S       >=
             W + R + F + S + N >=
END
```

where *M* represents the number of employees starting on Monday, *T* the number on Tuesday, and so on. Furthermore, all variables must have nonnegative integer values. The right-hand side values were omitted in this formulation, because they will be specified at runtime.

Staffing Example Using Visual C++

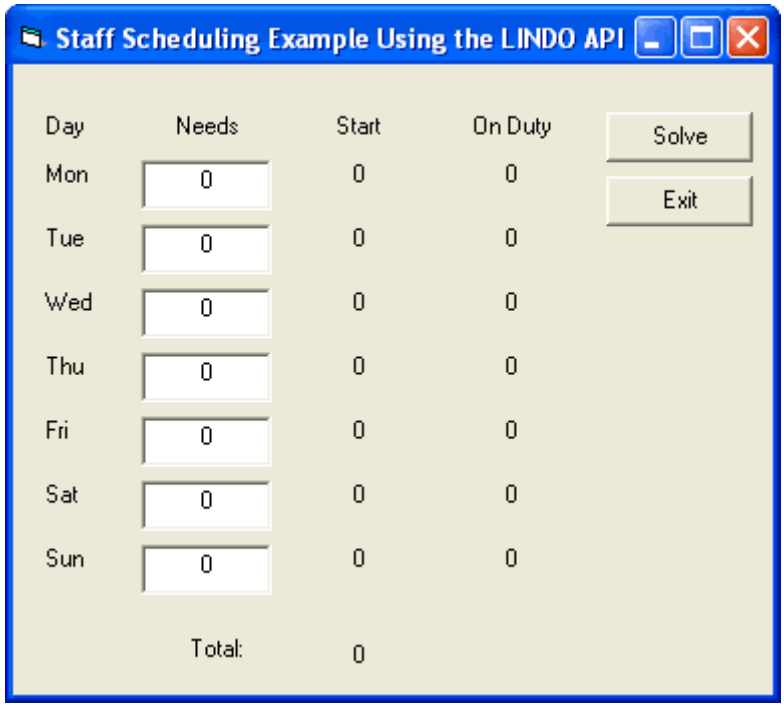
In this section, an application that interfaces with LINDO API to solve the Pluto Dogs problem will be built in Visual C++ 6.0. A complete version of this project may be found in \lindoapi\samples\c\samp3.

This example uses the MFC AppWizard in Visual C++ to build an MFC (Microsoft Foundation Class) Windows application for solving the Pluto Dogs problem. For those unfamiliar with MFC, it is an all-encompassing, object-oriented programming interface to Windows, designed for use with C++. MFC is not required to interface with LINDO API. It was chosen to use in our example because it greatly reduces the amount of development effort required to build a Windows application.

To build the sample application, start Visual C++ 6.0 and then follow these steps:

- Issue the *File|New* command.
- In the "New" dialog box, click on the "Projects" tab.
- On the "Projects" tab, click on the project type titled "MFC AppWizard (exe)", input a name for the project in the "Project Name" edit field, input the destination folder in the "Project Name" edit field, and click the *OK* button.
- You will see a dialog box titled "MFC AppWizard – Step 1". Click on the *Dialog Based* radio button, because our application will reside entirely within a single dialog box. Click the *Finish* button.
- Click the *OK* button to clear the "New Project Information" dialog, and the AppWizard will generate the skeleton code base for the application.

Next, modify the application's dialog box, so it appears as follows:



The user will input the staffing requirements in the "Needs" column. The application will read these requirements, and then build and solve the staffing integer programming model. To display the results, the application will place the optimal number of employees to start on each day of the week in the "Start" column, the number working each day in the "On Duty" column, and the total number of employees required in the "Total" field. The *Solve* button solves for the current staffing needs data, while the *Exit* button exits the application.

In order to access the various data fields in the dialog box, the ClassWizard in Visual C++ must be used to associate member variables with each of the data fields. After doing this, the handler code for the *Solve* button in the dialog class module should be edited, so that it is as follows:

```
#include "lindo.h"
#define APIERRORSETUP
   int nErrorCode;
   char cErrorMessage[LS_MAX_ERROR_MESSAGE_LENGTH]
/* Define a macro to do our error checking */
#define APIERRORCHECK
   if (nErrorCode)
   {
      if ( pEnv)
      {
         LSgetErrorMessage( pEnv, nErrorCode,
          cErrorMessage);
         printf("Errorcode=%d:  %s\n", nErrorCode,
          cErrorMessage);
         LSdeleteEnv( &pEnv);
      } else {
         printf( "Fatal Error\n");
      }
      return;
   }

void CStafflndDlg::OnSolve()
{
   APIERRORSETUP;
   pLSenv pEnv = NULL;
   char MY_LICENSE_KEY[1024];

// >>> Step 1 <<< Create an environment
   nErrorCode = LSloadLicenseString(
                "../../../license/lndapi60.lic",MY_LICENSE_KEY);
   APIERRORCHECK;
   pEnv = LScreateEnv( &nErrorCode, MY_LICENSE_KEY);
   if ( !pEnv)
   {
      AfxMessageBox("Unable to create environment!");
      return;
   }

// >>> Step 2 <<< Create a model in the environment
   pLSmodel pMod = NULL;
   pMod = LScreateModel( pEnv, &nErrorCode);
   APIERRORCHECK;

// >>> Step 3 <<< Construct the model
// Number of variables and constraints
   const int nVars = 7, nRows = 7;

// The direction of optimization
   int nDir = LS_MIN;

// The objective's constant term
   double dObjConst = 0.;
```

```
// The coefficients of the objective function
   double adC[ nVars] = {1.,1.,1.,1.,1.,1.,1.};
// Get right-hand sides of the constraints from
//  the Needs column of the dialog box
   UpdateData( true);
   double dNeeds[7];
   dNeeds[ 0] = m_nNeedsMon;
   dNeeds[ 1] = m_nNeedsTue;
   dNeeds[ 2] = m_nNeedsWed;
   dNeeds[ 3] = m_nNeedsThu;
   dNeeds[ 4] = m_nNeedsFri;
   dNeeds[ 5] = m_nNeedsSat;
   dNeeds[ 6] = m_nNeedsSun;

// The constraint types (all Greater-thans)
   char acConTypes[ nRows] = {'G','G','G','G','G','G','G'};

// The number of nonzeros in the constraint matrix
   const int nNZ = 35;

// The indices of the first nonzero in each column
   int anBegCol[ nVars + 1];
   for ( int i = 0; i <= nVars; i++)
   {
      anBegCol[ i] = 5 * i;
   }
// The length of each column. Since we aren't leaving
//  any blanks in our matrix, we can set this to NULL.
   int *pnLenCol = NULL;

// The nonzero coefficients and row indices
   double adA[ nNZ];
   int anRowX[ nNZ];
   int nX = 0;
   for ( i = 0; i < 7; i++)
   {
      for ( int j = i; j < i + 5; j++)
      {
          adA[ nX] = 1.;
          anRowX[ nX] = j % 7;
          nX++;
      }
   }
// Simple upper and lower bounds on the variables.
//  By default, all variables have a lower bound of zero
//  and an upper bound of infinity.  Therefore pass NULL
//  pointers in order to use these default values.
   double *pdLower = NULL, *pdUpper = NULL;

// We have now assembled a full description of the model.
//  We pass this information to LSloadLPData with the
//  following call.
   nErrorCode = LSloadLPData( pMod, nVars, nRows, nDir,
    dObjConst, adC, dNeeds, acConTypes, nNZ, anBegCol,
    pnLenCol, adA, anRowX, pdLower, pdUpper);
   APIERRORCHECK;
```

```
// Mark all 7 variables as being general integer
   nErrorCode = LSloadMIPData( pMod, "IIIIIII");
   APIERRORCHECK;

// >>> Step 4 <<< Perform the optimization
   nErrorCode = LSsolveMIP( pMod, NULL);
   APIERRORCHECK;

// >>> Step 5 <<< Retrieve the solution
   double dObjVal, dStart[ 7], dSlacks[ 7];
   nErrorCode = LSgetInfo(pMod, LS_DINFO_MIP_OBJ, &dObjVal);
   APIERRORCHECK;
   nErrorCode = LSgetMIPPrimalSolution( pMod, dStart);
   APIERRORCHECK;
   nErrorCode = LSgetMIPSlacks( pMod, dSlacks);
   APIERRORCHECK;

// Display solution in dialog box
   m_csTotal.Format( "%d", (int) dObjVal);
   m_csStartMon.Format( "%d", (int) dStart[ 0]);
   m_csStartTue.Format( "%d", (int) dStart[ 1]);
   m_csStartWed.Format( "%d", (int) dStart[ 2]);
   m_csStartThu.Format( "%d", (int) dStart[ 3]);
   m_csStartFri.Format( "%d", (int) dStart[ 4]);
   m_csStartSat.Format( "%d", (int) dStart[ 5]);
   m_csStartSun.Format( "%d", (int) dStart[ 6]);
   m_csOnDutyMon.Format( "%d", (int) ( dNeeds[ 0] - dSlacks[ 0]));
   m_csOnDutyTue.Format( "%d", (int) ( dNeeds[ 1] - dSlacks[ 1]));
   m_csOnDutyWed.Format( "%d", (int) ( dNeeds[ 2] - dSlacks[ 2]));
   m_csOnDutyThu.Format( "%d", (int) ( dNeeds[ 3] - dSlacks[ 3]));
   m_csOnDutyFri.Format( "%d", (int) ( dNeeds[ 4] - dSlacks[ 4]));
   m_csOnDutySat.Format( "%d", (int) ( dNeeds[ 5] - dSlacks[ 5]));
   m_csOnDutySun.Format( "%d", (int) ( dNeeds[ 6] - dSlacks[ 6]));
   UpdateData( false);

// >>> Step 6 <<< Delete the LINDO environment
   LSdeleteEnv( &pEnv);
}
```

Prior to the point where the application begins constructing the model, the code should be familiar and require no explanation. Construction of the model is begun with the following code:

```
// >>> Step 3 <<< Construct the model

// Number of variables and constraints
   const int nVars = 7, nRows = 7;

// The direction of optimization
   int nDir = LS_MIN;

// The objective's constant term
   double dObjConst = 0.;

// The coefficients of the objective function
   double adC[ nVars] = {1.,1.,1.,1.,1.,1.,1.};
```

There are seven decision variables in this model — one for each day of the week to determine the number of employees to start on each day. There are also seven constraints — one for each day of the week to insure that the number of staff on duty on each day exceeds the specified staffing requirements. The objective in this example is to minimize the total number of employees hired. Thus, the direction of the objective is LS_MIN. There is no constant term in the objective function, so it is set to 0. The total number of employees in the objective must be summed. Thus, we place a coefficient of 1 on each of the seven variables in the objective row.

Next, the staffing requirements is loaded from the dialog box into an array:

```
// Get right-hand sides of the constraints from
//  the Needs column of the dialog box
   UpdateData( true);
   double dNeeds[7];
   dNeeds[ 0] = m_nNeedsMon;
   dNeeds[ 1] = m_nNeedsTue;
   dNeeds[ 2] = m_nNeedsWed;
   dNeeds[ 3] = m_nNeedsThu;
   dNeeds[ 4] = m_nNeedsFri;
   dNeeds[ 5] = m_nNeedsSat;
   dNeeds[ 6] = m_nNeedsSun;
```

This array will be passed to LINDO as the array of right-hand side values.

Each of the seven constraints are of the form total staffing must be greater-than-or-equal-to staffing requirements. So, a string of seven uppercase letter G's is constructed to indicate all the constraints are of type greater-than-or-equal-to:

```
// The constraint types (all Greater-thans)
   char acConTypes[ nRows] = {'G','G','G','G','G','G','G'};
```

Each column in the model has five nonzero coefficients of 1, representing the five days of the week worked. Thus, given that there are seven columns, there are a total of 35 nonzero coefficients:

```
// The number of nonzeros in the constraint matrix
   const int nNZ = 35;
```

Since there are 5 nonzeros per column, the column-starting pointers are 0, 5, 10, 15, 20, 25, 30, and 35:

```
// The indices of the first nonzero in each column */
   int anBegCol[ nVars + 1];
   for ( int i = 0; i <= nVars; i++)
   {
      anBegCol[ i] = 5 * i;
   }
```

Note that an eighth column-starting pointer that points to the position immediately following the last nonzero must be defined.

We are passing LINDO a dense array of nonzeros, so the column lengths can be inferred from the column-starting pointers. Thus, the column-length pointer can be set to NULL:

```
// The length of each column. Since we aren't leaving
//  any blanks in our matrix, we can set this to NULL.
   int *pnLenCol = NULL;
```

The next code segment generates the nonzero coefficients of the constraints and is a little tricky:

```
// The nonzero coefficients and row indices
   double adA[ nNZ];
   int anRowX[ nNZ];

   int nX = 0;
   for ( i = 0; i < 7; i++)
   {
      for ( int j = i; j < i + 5; j++)
      {
          adA[ nX] = 1.;
          anRowX[ nX] = j % 7;
          nX++;
      }
   }
```

A double loop is used here. The outer loop runs *i* from 0 to 6, indexing over the seven columns that are generated. In the inner loop, 5 nonzeros of value 1 are generated representing the five days worked for the column. The column representing employees starting on Monday will have nonzeros in rows 0 through 4, representing the Mon – Fri work schedule. Rows 5 and 6 will not have coefficients due to the fact that Monday starters are off Saturday and Sunday. Things get a little more complicated later in the week. Suppose the nonzeros for the Thursday starters are being generated. These occur in the Thu, Fri, Sat, Sun, and Mon rows. The problem comes when the schedule needs to "wrap" around from Sunday to Monday. This is done by using the modulo operator (%), which wraps any row index of 7, or higher, around to the start of the week. A picture of the nonzero matrix for this model would appear as follows:

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Obj	1	1	1	1	1	1	1
Mon	1			1	1	1	1
Tue	1	1			1	1	1
Wed	1	1	1			1	1
Thu	1	1	1	1			1
Fri	1	1	1	1	1		
Sat		1	1	1	1	1	
Sun			1	1	1	1	1

Each column has a contiguous block of 5 nonzero coefficients. In each subsequent column, the block is shifted down one row. Starting with Thursday's column, one or more nonzeros must wrap back to the top.

The default bounds of zero to infinity are accepted by setting the bounds pointers to NULL:

```
// Simple upper and lower bounds on the variables.
//  By default, all variables have a lower bound of zero
//  and an upper bound of infinity. Therefore pass NULL
//  pointers in order to use these default values.
   double *pdLower = NULL, *pdUpper = NULL;
```

The model has now been generated, so it will be passed to LINDO API by calling *LSloadLPData*():

```
// We have now assembled a full description of the model.
//  We pass this information to LSloadLPData with the
//  following call.
   nErrorCode = LSloadLPData( pMod, nVars, nRows, nDir,
    dObjConst, adC, dNeeds, acConTypes, nNZ, anBegCol,
    pnLenCol, adA, anRowX, pdLower, pdUpper);
   APIERRORCHECK;
```

Up to this point, nothing has been indicated to LINDO API regarding the integrality requirement on the variables. We do this through a call to *LSloadVarType*():

```
// Mark all 7 variables as being general integer
   nErrorCode = LSloadVarType( pMod, "IIIIIII");
   APIERRORCHECK;
```

Each of the seven variables are integer, which is indicated by passing a string of seven letter *I*'s. Note that *LSloadVarType*() must be called *after LSloadLPData*(). Attempting to call *LSloadVarType*() prior to the call to *LSloadLPData*() will result in an error.

The next step is to solve the model:

```
// >>> Step 4 <<< Perform the optimization
   nErrorCode = LSsolveMIP( pMod, NULL);
   APIERRORCHECK;
```

In this case, the branch-and-bound solver must be called with *LSsolveMIP*(), because we have integer variables in our model.

Next, the solution values are retrieved:

```
// >>> Step 5 <<< Retrieve the solution
   double dObjVal, dStart[ 7], dSlacks[ 7];
   nErrorCode = LSgetInfo(pMod, LS_DINFO_MIP_OBJ, &dObjVal);
   APIERRORCHECK;
   nErrorCode = LSgetMIPPrimalSolution( pMod, dStart);
   APIERRORCHECK;
   nErrorCode = LSgetMIPSlacks( pMod, dSlacks);
   APIERRORCHECK;
```

Note that the query routines that are specifically designed for MIP models have been used.
The remainder of the code is straightforward and deals with posting the solution in the dialog box and freeing the LINDO environment.

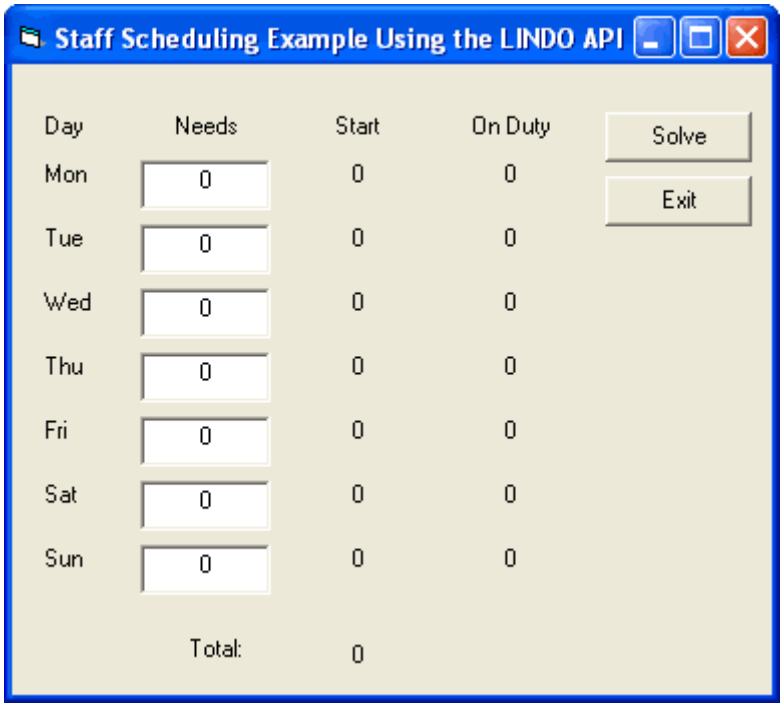
Staffing Example Using Visual Basic

This section will construct an example using the Visual Basic 6.0 development environment.

After starting VB 6.0, a new project can be created using the following steps:

- Run the *File|New Project* command.
- In the "New Project" dialog box, click once on the "Standard EXE" icon, then press the *OK* button.

A simple form for the application's dialog box will be presented. Modify this form, so it appears as follows:



Double click on the *Solve* button in the form and enter the code below:

```
Private Sub Solve_Click()

Dim nErr As Long
Dim pEnv As Long
Dim LicenseKey As String * LS_MAX_ERROR_MESSAGE_LENGTH

nErr = LSloadLicenseString("..\..\..\license\lndapi60.lic",
       LicenseKey)
Call CheckErr(pEnv, nErr)

'>>> Step 1 <<<:  Create a LINDO environment.
pEnv = LScreateEnv(nErr, LicenseKey)
If (nErr > 0) Then
   MsgBox ("Unable to create environment.")
   End
End If

'>>> Step 2 <<< create a model in the environment
Dim pMod As Long
pMod = LScreateModel(pEnv, nErr)
Call CheckErr(pEnv, nErr)

'>>> Step 3 <<< construct the model
'number of variables
Dim nVars As Long
```

```
nVars = 7
'number of constraints
Dim nRows As Long
nRows = 7
'direction of objective
Dim nDir As Long
nDir = LS_MIN
'objective constant term
Dim dObjConst As Double
dObjConst = 0
'objective coefficients
ReDim dObjCoef(nVars) As Double
Dim i As Integer
For i = 0 To nVars - 1
   dObjCoef(i) = 1
Next
'get the staffing needs for the model's right-hand sides
ReDim dB(nVars) As Double
For i = 0 To nVars - 1
   dB(i) = Needs(i)
Next
'define the constraint types
Dim cConTypes As String
For i = 0 To nRows - 1
  cConTypes = cConTypes & "G"
Next
'the number of nonzero coefficients
Dim nNZ As Long
nNZ = 35
'the array of column start indices
ReDim nBegCol(nVars + 1) As Long
For i = 0 To nVars
   nBegCol(i) = 5 * i
Next
'the nonzero coefficients
ReDim dA(nNZ) As Double
ReDim nRowX(nNZ) As Long
Dim j, k As Integer
k = 0
For i = 0 To nVars - 1
  For j = 0 To 4
    nRowX(k) = (j + i) Mod 7
    dA(k) = 1
    k = k + 1
  Next j
Next i
'load the problem
nErr = LSloadLPData(pMod, nRows, nVars, nDir, _
 dObjConst, dObjCoef(0), dB(0), cConTypes, nNZ, _
 nBegCol(0), ByVal 0, dA(0), nRowX(0), ByVal 0, _
 ByVal 0)
Call CheckErr(pEnv, nErr)
'integer restrictions on the variables
Dim cVarType As String
For i = 1 To nVars
  cVarType = cVarType & "I"
```

```
Next
nErr = LSloadVarType(pMod, cVarType)
Call CheckErr(pEnv, nErr)

'>>> Step 4 <<< solve the model
nErr = LSsolveMIP(pMod, ByVal 0)
Call CheckErr(pEnv, nErr)

'>>> Step 5 <<< retrieve the solution
ReDim dX(nVars) As Double
Dim dObj As Double
Dim dSlacks(7) As Double
nErr = LSgetInfo(pMod, LS_DINFO_MIP_OBJ, dObj)
Call CheckErr(pEnv, nErr)
nErr = LSgetMIPPrimalSolution(pMod, dX(0))
Call CheckErr(pEnv, nErr)
nErr = LSgetMIPSlacks(pMod, dSlacks(0))
Call CheckErr(pEnv, nErr)
'post solution in dialog box
Total = dObj
For i = 0 To nVars - 1
   OnDuty(i) = dB(i) - dSlacks(i)
   Start(i) = dX(i)
Next

'>>> Step 6 <<< Delete the LINDO environment
Call LSdeleteEnv(pEnv)

End Sub

Public Sub CheckErr(pEnv As Long, nErr As Long)
' Checks for an error condition.  If one exists, the
'  error message is displayed then the application
'  terminates.
   If (nErr > 0) Then
      Dim cMessage As String
      cMessage = String(LS_MAX_ERROR_MESSAGE_LENGTH,
       _vbNullChar)
      Call LSgetErrorMessage(pEnv, nErr, cMessage)
      MsgBox (cMessage)
      End
   End If
End Sub
```

Prior to the point where construction of the model begins, the code should be familiar and require no explanation. Construction of the model begins with the following code:

```
'>>> Step 3 <<< construct the model
'number of variables
Dim nVars As Long
nVars = 7
'number of constraints
Dim nRows As Long
nRows = 7
'direction of objective
Dim nDir As Long
```

```
nDir = LS_MIN
'objective constant term
Dim dObjConst As Double
dObjConst = 0
'objective coefficients
ReDim dObjCoef(nVars) As Double
Dim i As Integer
For i = 0 To nVars - 1
   dObjCoef(i) = 1
Next
```

There are seven decision variables in this model – one for each day of the week to determine the number of employees to start on each day. There are also seven constraints – one for each day of the week to insure that the number of staff on duty on each day exceeds the specified staffing requirements. The objective is to minimize the total number of employees hired. Thus, the direction of the objective is LS_MIN. There is no constant term in the objective function, so it is set to 0. The total number of employees in the objective must be summed. Thus, a coefficient of 1 is placed on each of the seven variables in the objective row.

Next, the staffing requirements are loaded from the dialog box into an array:

```
'get the staffing needs for the model's right-hand sides
ReDim dB(nVars) As Double
For i = 0 To nVars - 1
   dB(i) = Needs(i)
Next
```

This array will be passed to LINDO API as the array of right-hand side values.

Each of the seven constraints are of the form total staffing must be greater-than-or-equal-to staffing requirements. So, a string of seven uppercase letter G's is constructed to indicate all the constraints are of type greater-than-or-equal-to:

```
'define the constraint types
Dim cConTypes As String
For i = 0 To nRows - 1
   cConTypes = cConTypes & "G"
Next
```

Each column in the model has five nonzero coefficients of 1, representing the five days of the week worked. Thus, given that there are seven columns, there are a total of 35 nonzero coefficients:

```
'the number of nonzero coefficients
Dim nNZ As Long
nNZ = 35
```

Since there are 5 nonzeros per column, the column-starting pointers are 0, 5, 10, 15, 20, 25, 30, and 35:

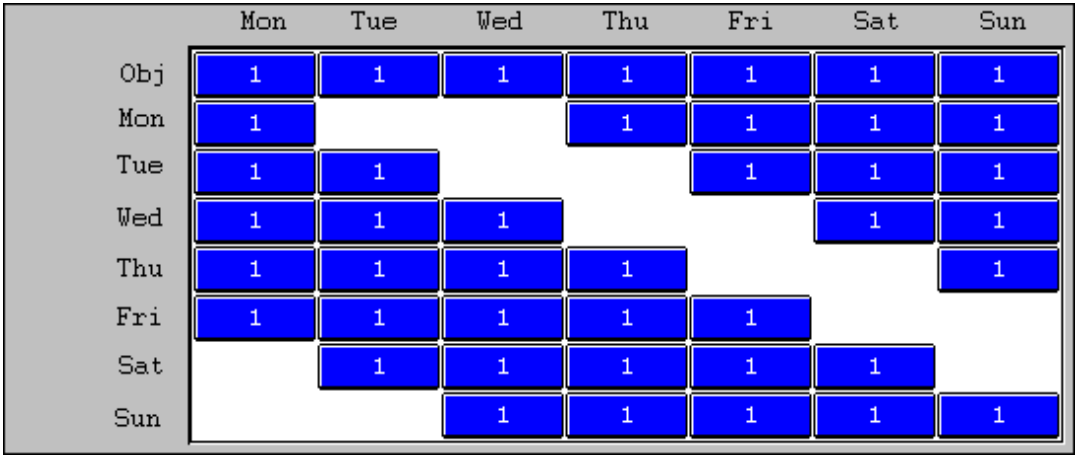
```
'the array of column start indices
ReDim nBegCol(nVars + 1) As Long
For i = 0 To nVars
   nBegCol(i) = 5 * i
Next
```

Note that an eighth column-starting pointer that points to the position immediately following the last nonzero must be defined.

The next code segment generates the nonzero coefficients of the constraints and is a little tricky:

```
'the nonzero coefficients
ReDim dA(nNZ) As Double
ReDim nRowX(nNZ) As Long
Dim j, k As Integer

k = 0
For i = 0 To nVars - 1
  For j = 0 To 4
    nRowX(k) = (j + i) Mod 7
    dA(k) = 1
    k = k + 1
  Next j
Next i
```

A double loop is used here. The outer loop runs *i* from 0 to 6, indexing over the seven columns that are generated. In the inner loop, 5 nonzeros of values 1 are generated representing the five days worked for the column. The column representing employees starting on Monday will have nonzeros in rows 0 through 4, representing the Mon – Fri work schedule. Rows 5 and 6 will not have coefficients due to the fact that Monday starters are off Saturday and Sunday. Things get a little more complicated later in the week. Suppose the nonzeros for the Thursday starters are being generated. These occur in the Thu, Fri, Sat, Sun, and Mon rows. The problem comes when the schedule must "wrap" around from Sunday to Monday. This is done by using the modulo operator (mod), which wraps any row index of 7, or higher, around to the start of the week. A picture of the nonzero matrix for this model would appear as follows:



Each column has a contiguous block of 5 nonzero coefficients in the constraints. In each subsequent column, the block is shifted down one row. Starting with Thursday's column, one or more nonzeros must wrap back to the top.

The model has now been generated, so it may be passed to LINDO API by calling *LSloadLPData*():

```
'load the problem
nErr = LSloadLPData(pMod, nRows, nVars, nDir, _
 dObjConst, dObjCoef(0), dB(0), cConTypes, nNZ, _
 nBegCol(0), ByVal 0, dA(0), nRowX(0), ByVal 0, _
 ByVal 0)
Call CheckErr(pEnv, nErr)
```

Note that three of the arguments are set to *ByVal 0*, which indicates those arguments are being omitted and their default values accepted. The first of these arguments is for the array of column lengths. Since the nonzero matrix includes no blank spaces, the column-length array is redundant. The remaining two 0 arguments are for the variable bound arrays. These are omitted because the default variable bound of zero to infinity is correct for this model.

After the call to *LSloadLPData*(), a test is done to see if any error condition was raised by calling our *CheckErr*() routine. *CheckErr*() should be familiar from the discussions in the previous chapter.

Up to this point, nothing has been indicated to LINDO API regarding the integrality requirement on the variables. This is done through a call to *LSloadVarType*():

```
'integer restrictions on the variables
Dim cVarType As String
For i = 1 To nVars
  cVarType = cVarType & "I"
Next
nErr = LSloadVarType(pMod, cVarType)
Call CheckErr(pEnv, nErr)
```

Each of the seven variables are integer, which is indicated by passing a string of seven letter *I*'s. Note that *LSloadVarType*() must be called *after LSloadLPData*(). Attempting to call *LSloadVarType*() prior to the call to *LSloadLPData*() will result in an error.

The next step is to solve the model:

```
'>>> Step 4 <<< solve the model
nErr = LSsolveMIP(pMod, ByVal 0)
Call CheckErr(pEnv, nErr)
```

In this case, the branch-and-bound solver must be called with *LSsolveMIP*(), because there are integer variables in our model.

Next, the solution values are retrieved:

```
'>>> Step 5 <<< retrieve the solution
ReDim dX(nVars) As Double
Dim dObj As Double
Dim dSlacks(7) As Double
nErr = LSgetInfo(pMod, LS_DINFO_MIP_OBJ, dObj)
Call CheckErr(pEnv, nErr)
nErr = LSgetMIPPrimalSolution(pMod, dX(0))
Call CheckErr(pEnv, nErr)
nErr = LSgetMIPSlacks(pMod, dSlacks(0))
Call CheckErr(pEnv, nErr)
'post solution in dialog box
Total = dObj
For i = 0 To nVars - 1
   OnDuty(i) = dB(i) - dSlacks(i)
   Start(i) = dX(i)
Next
```

Note that the query routines that are specifically designed for MIP models have been used.

The remainder of the code is straightforward and deals with posting the solution in the dialog box and deleting the LINDO environment.

Chapter 5: Solving Quadratic Programs

The quadratic programming interface of LINDO API is designed to solve quadratically constrained problems (QCP) of the form:

Optimize $\frac{1}{2} x'Q^c x + cx$;
subject to:
$\frac{1}{2} x'Q^i x + a_i x \; ? \; b_i$ for $i = 0,1,\ldots,m-1$,
$L_j \le x_j \le U_j$ for $j = 0,1,\ldots,n-1$,
x_j is integer for j in a specified $J \subseteq \{0,\ldots, n-1\}$

where

Optimize is either minimize or maximize,
Q^c, and Q^i are symmetric n by n matrices of constants for $i=0,\ldots,m-1$,
c and a_i are 1 by n vectors of constants,
$x = \{x_0, x_2,\ldots,x_{n-1}\}$, is an n-vector of decision variables.
"?" is one of the relational operators "≤", "=", or "≥".

LINDO API will solve and return a global optimum if:

Q^c is positive semi-definite for a minimize objective, or,
Q^c is negative semi-definite for a maximize objective, and
Q^i is positive semi-definite for each ≤ constraint, and
Q^i is negative semi-definite for each ≥ constraint.

All the above are also true if "semi-" is deleted. LINDO API may not find a global optimum if some Q is indefinite, or some constraint with a Q on the left-hand side is an equality constraint. If, for example, Q is a valid covariance matrix, then Q is positive semi-definite. The negative of a positive semi-definite matrix is negative semi-definite, and vice versa. If Q is positive (negative) semi-definite, then all of its eigen values are non-negative (non-positive). Strictly positive definite implies that Q is full rank. If the Q^i matrices are positive (negative) semi-definite for "≤" ("≥") type constraints and equal to zero for "=" type constraints, then the feasible region is convex. Geometrically, a positive definite matrix corresponds to a bowl shaped function. A positive semi-definite matrix corresponds to a trough shaped function. A negative definite matrix corresponds to an umbrella shaped function.

The ½ term is used above for historical reasons related to the fact that the derivative of $\frac{1}{2} x'Qx$ is Qx.

Note: LINDO API uses the barrier algorithm to solve quadratic programs when they satisfy the semi-definiteness conditions listed above. Otherwise, the nonlinear solver will be used. In the latter case, the global optimality cannot be ensured unless the global optimization is performed.

Setting up Quadratic Programs

There are three ways by which you can setup a quadratic program with LINDO API. The first one is reading a quadratic program directly from an MPS format file, using an extended format to incorporate quadratic forms in the model formulation. The second way is to build the model directly in memory and pass the data of the matrices representing quadratic forms to LINDO API. The third way is to formulate the model using an instruction list (as described in Chapter 7 and Appendix D). Here, we describe the first and second ones.

Loading Quadratic Data via Extended MPS Format Files

The quadratic parts of the objective function and the constraints can be described in an MPS file by adding a *QMATRIX* section to the file for each quadratic form. Synonyms that can be used for this section are *QSECTION* or *QUADS*. The row labels that follow the QMATRIX term denote the constraints the quadratic terms belong to. The following example illustrates how an MPS file is modified to include quadratic terms in the objective function.

Example:

Suppose the quadratic program under consideration is:

```
Minimize 0.5*(     X0*X0 +  .75*X0*X1
               0.75*X0*X1 + 2.00*X1*X1) + X0 + X1;
Subject to:
                X0 + X1 >= 10;
```

The corresponding MPS file to this quadratic program is:

```
NAME          quadex2
ROWS
 N  OBJ
 G  C1
COLUMNS
    X0        OBJ         1.0
    X0        C1          1.0
    X1        OBJ         1.0
    X1        C1          1.0
RHS
    RHS       C1          10.
QMATRIX       OBJ
    X0        X0          1.0
    X0        X1          0.75
    X1        X1          2.0
ENDATA
```

The format of the *QMATRIX* section is similar to the *COLUMNS* section except that the first two columns on each line correspond to a pair of variables for which their product appears as a term in the quadratic objective and the third column on a line corresponds to the coefficient of this product. The presence of the factor 0.5 is assumed when specifying these coefficients. In describing the *QMATRIX*, it is sufficient to specify the elements on its diagonal and below-diagonal entries because the quadratic matrices are assumed to be symmetric. It should be noted that only one *QMATRIX* section is allowed for each constraint and no *QMATRIX* sections can precede the *COLUMNS* section.

The solution file for the above example will contain the report:

```
PROBLEM NAME     quadex2
 QP OPTIMUM FOUND

 ITERATIONS BY SIMPLEX METHOD =           0
 ITERATIONS BY BARRIER METHOD =           6
 ITERATIONS BY NLP METHOD     =           0
 TIME ELAPSED (s)             =           0

 OBJECTIVE FUNCTION VALUE
  1)               57.916666753

 VARIABLE                VALUE                REDUCED COST
  X0                       8.333333307             0.000000010
  X1                       1.666666701             0.000000060

 ROW               SLACK OR SURPLUS            DUAL PRICES
  C1                       -0.000000008            10.583333322

END OF REPORT
```

Note: Your license must have the barrier or nonlinear license options to be able to work with quadratic formulations. Attempting to solve a problem that has a quadratic objective or constraint using other optimization algorithms such as primal simplex, dual simplex, or mixed-integer solver will return an error.

Loading Quadratic Data via API Functions

The second way to input a QCP is by setting-up a problem structure and using LINDO API's quadratic programming functions to specify the quadratic terms. In this framework, your front-end program should perform at least the following steps to enter the problem and retrieve its solution:

- Create a LINDO environment with a call to *LScreateEnv*().
- Create a model structure in this environment with a call to *LScreateModel*().
- Load problem structure and linear data into the model structure with a call to *LSloadLPData*().
- Load the quadratic problem data into the model structure with a call to *LSloadQCData*().
- Load (optionally) the integer-programming data with a call to *LSloadVarType*().
- Solve the problem with a call to *LSoptimize*() (or *LSsolveMIP*() if there are integer variables).
- Retrieve the solution with calls to *LSgetInfo*(), *LSgetPrimalSolution*(), and *LSgetDualSolution*().
- Delete the model and environment with a call to *LSdeleteEnv*().

The step specific to loading quadratic models is Step 4. Quadratic terms in each row, as well as the objective function, are represented with a symmetric matrix. Each of these matrices is described by a vector of four-tuples or quadruplets, one quadruplet per nonzero. Each quadruplet contains:

- index of the constraint which the quadratic matrix belongs,
- row index *i* (actually the index of a column) of the nonzero in quadratic matrix,
- column index *j* of the nonzero in quadratic matrix,
- nonzero value *q(i,j)*.

We illustrate the preparation of the data with an example:

$$\begin{array}{lll} \text{Maximize} & 3x_0 + 10x_1 - 2\,x_0^2 - 3x_1^2 \\ & - 4x_2^2 + 2\,x_0x_2 + 5x_2x_1 \\ \text{s.t.} & \\ \text{Constraint 0:} & (x_0 - 1)^2 + (x_1 - 1)^2 \le 1 \\ \text{Constraint 1:} & (x_1 - 3)^2 + (x_2 - 1)^2 \le 2 \end{array}$$

$$\begin{array}{lcccl} -\infty & \le & x_0 & \le & +\infty \\ -\infty & \le & x_1 & \le & +\infty \\ -\infty & \le & x_2 & \le & +\infty \end{array}$$

This model can be written in the equivalent symmetric matrix form

$$\begin{array}{ll} \text{Maximize} & 3\,x_0 + 10x_1 + \\ & \tfrac{1}{2}(-4\,x_0^2 + 2x_0x_2 \\ & \quad -6\,x_1^2 + 5x_1x_2 \\ & \quad + 2x_2x_0 + 5x_2x_1 - 8\,x_2^2) \\ \text{s.t.} & \\ \text{Constraint 0:} & -\,x_0 - x_1 + \tfrac{1}{2}(2\,x_0^2 + 2\,x_1^2) \le -1 \\ \text{Constraint 1:} & -3\,x_1 - x_2 + \tfrac{1}{2}(2\,x_1^2 + 2\,x_2^2) \le -4 \end{array}$$

$$\begin{array}{lcccl} -\infty & \le & x_0 & \le & +\infty \\ -\infty & \le & x_1 & \le & +\infty \\ -\infty & \le & x_2 & \le & +\infty \end{array}$$

Digression: The historic reason for writing the quadratic part in this form, with the factor of 1/2 in front, is as follow. When first partial derivatives are taken, the 1/2 cancels out, and the coefficients of the linear first order conditions that the computer solves are exactly the coefficients inside the parentheses.

Several other conventions of note are: a) the LINDO API numbers the constraints starting at 0, 1, …, b) the objective row is denoted as row -1, and c) because of symmetry, we only input the upper triangle of the symmetric matrix. Thus, the equivalents of the above matrices in quadruplet form are:

$Q^{obj} =$

	x_0	x_1	x_2
x_0	-4	0	2
x_1	0	-6	5
x_2	2	5	-8

⇒

Constraint Index	Row index	Column index	Nonzero value
-1	0	0	-4
-1	0	2	2
-1	1	1	-6
-1	1	2	5
-1	2	2	-8

And those associated with constraints 0 and 1 are Q^0 and Q^1, with 2 nonzeros in each.

$Q^0 =$

	x_0	x_1	x_2
x_0	2	0	0
x_1	0	2	0
x_2	0	0	0

⇒

Constraint Index	Row index	Column index	Nonzero value
0	0	0	2
0	1	1	2

$Q^1 =$

	x_0	x_1	x_2
x_0	0	0	0
x_1	0	2	0
x_2	0	0	2

⇒

Constraint Index	Row index	Column index	Nonzero value
1	1	1	2
1	2	2	2

Combining the quadruplet representations of Q^{obj}, Q^0 and Q^1, we obtain the following arrays:

Constraint Index	Row index	Column index	Nonzero value
-1	0	0	-4
-1	0	2	2
-1	1	1	-6
-1	1	2	5
-1	2	2	-8
0	0	0	2
0	1	1	2
1	1	1	2
1	2	2	2

The quadratic data for this model is now ready to be loaded to the solver. Using C conventions, the following code fragment sets up the arrays and then calls the LSloadQCData function to load these four vectors into the LINDO API. The LP data must have been previously loaded using LSloadLPData.

```
{
  pLSmodel pModel;
  int nQCnnz = 9;
  int paiQCrows [9] = {  -1,  -1,  -1,  -1,  -1,   0,   0,   1,   1};
  int paiQCcols1[9] = {   0,   0,   1,   1,   2,   0,   1,   1,   2};
  int paiQCcols2[9] = {   0,   2,   1,   2,   2,   0,   1,   1,   2};
  int padQCcoef [9] = {-4.0, 2.0,-6.0, 5.0,-8.0, 2.0, 2.0, 2.0, 2.0};
  int nErr;

  nErr = LSloadQCData(pModel, nQCnnz, paiQCrows, paiQCcols1,
    paiQCcols2, padQCcoef);
}
```

We recommend that you load only the upper triangular portion of a *Q* matrix when you call LSloadQCData. You can in fact load the lower triangular portion of the matrix, or even the full

matrix, and the matrix need not be symmetric. If LSloadQCData finds one or more nonzero instances of the matrix element q_{ij} or q_{ji}, it treats both q_{ij} and q_{ji} as equal to the average of all the elements supplied for q_{ij} and q_{ji}. This, for example allows you to supply an asymmetric Q matrix and LSLoadQCData will automatically convert it to the equivalent symmetric matrix.

In the following examples, the functions in LINDO API that are related to solving quadratic problems are described.

Sample Portfolio Selection Problems

A common use of quadratic programs is in portfolio selection in finance where the proportion of the available assets invested in each investment alternative is determined. The following examples illustrate the use of LINDO API to build and solve small portfolio selection models.

Example 1. The Markowitz Model:

Consider a portfolio problem with n assets or stocks held over one period. Let w_i denote the amount of asset i invested and held throughout the period, and r_i denote the return of asset i over the period. The decision variable is the vector w with two basic assumptions: $w_i \geq 0$ (short positions are not allowed) and $w_1 + w_2 + \ldots + w_n = 1$ (i.e., unit total budget).

This example assumes the investor wishes to use the well known Markowitz model to balance the average expected risk and average return on each dollar invested in selecting the portfolio. This can be handled by maximizing the expected return while limiting the risk of loss with a constraint of the form $w'Q\,w \leq K$. Here, Q is the covariance matrix of returns and K is a bound on the risk of loss.

The following C programming code illustrates how this model can be set up and solved using LINDO API for a small portfolio selection problem.

```
 /*

#####################################################################
#                       LINDO-API
#                    Sample Programs
#                  Copyright (c) 2007 by LINDO Systems, Inc
#
#         LINDO Systems, Inc.           312.988.7422
#         1415 North Dayton St.         info@lindo.com
#         Chicago, IL 60622             http://www.lindo.com
#####################################################################

  File   : markow.c
  Purpose: Solve a quadratic programming problem.
  Model  : The Markowitz Portfolio Selection Model

           MAXIMIZE  r(1)w(1) + ... +r(n)w(n)
           st.       sum_{ij} Q(i,j)w(i)w(j) <= K
                         w(1) + ..... + w(n)  = 1
                         w(1),          ,w(n) >= 0
           where
           r(i)   : return on asset i
           Q(i,j): covariance between the returns of i^th and
                   j^th assets.
```

```
            K     : a scalar denoting the level of risk of loss.
            w(i)  : proportion of total budget invested on asset i

  Covariance Matrix:
          w1    w2    w3    w4
     w1 [ 1.00  0.64  0.27  0.    ]
     w2 [ 0.64  1.00  0.13  0.    ]
     w3 [ 0.27  0.13  1.00  0.    ]
     w4 [ 0.    0.    0.    1.00  ]

  Returns Vector:
          w1    w2    w3    w4
  r =   [ 0.30  0.20 -0.40  0.20  ]

  Risk of Loss Factor:
  K = 0.4

*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "lindo.h"

/* Define a macro to declare variables for
    error checking */
#define APIERRORSETUP
   int nErrorCode;
   char cErrorMessage[LS_MAX_ERROR_MESSAGE_LENGTH]

/* Define a macro to do our error checking */
#define APIERRORCHECK
   if (nErrorCode)
   {
      if ( pEnv)
      {
         LSgetErrorMessage( pEnv, nErrorCode,
          cErrorMessage);
         printf("Errorcode=%d:  %s\n", nErrorCode,
          cErrorMessage);
      } else {
         printf( "Fatal Error\n");
      }
      exit(1);
   }

/* main entry point */
int main(int argc, char **argv)
{
   APIERRORSETUP;
   int nM = 2;      /* Number of constraints */
   int nN = 4;      /* Number of assets */

   double K = 0.20; /* 1/2 of the risk level*/

  /* declare an instance of the LINDO environment object */
   pLSenv pEnv = NULL;
```

```
/* declare an instance of the LINDO model object */
 pLSmodel pModel;

 char MY_LICENSE_KEY[1024];

/*************************************************************
 * Step 1: Create a model in the environment.
 *************************************************************/
 nErrorCode = LSloadLicenseString(
              "../../../license/lndapi60.lic",MY_LICENSE_KEY);

 pEnv = LScreateEnv ( &nErrorCode, MY_LICENSE_KEY);
 if ( nErrorCode == LSERR_NO_VALID_LICENSE)
 {
    printf( "Invalid License Key!\n");
    exit( 1);
 }
 APIERRORCHECK;

/*************************************************************
 * Step 2: Create a model in the environment.
 *************************************************************/
 pModel = LScreateModel ( pEnv, &nErrorCode);
 APIERRORCHECK;

/*************************************************************
 * Step 3: Specify and load the LP portion of the model.
 *************************************************************/
 {
   /* The direction of optimization */
    int objsense = LS_MAX;
    /* The objective's constant term */
    double objconst = 0.;
    /* The coefficients of the objective function are the expected
    returns*/
    double reward[4] = { .3, .2, -.4, .2};
    /* The right-hand sides of the constraints */
    double rhs[2] = { K, 1.0 };
    /* The constraint types */
    char contype[2] = {'L','E'};
    /* The number of nonzeros in the constraint matrix */
    int Anz = 4;
    /* The indices of the first nonzero in each column */
    int Abegcol[5] = { 0, 1, 2, 3, Anz};
    /* The length of each column.  Since we aren't leaving
     * any blanks in our matrix, we can set this to NULL */
    int *Alencol = NULL;
    /* The nonzero coefficients */
    double A[4] = { 1., 1., 1., 1.};
    /* The row indices of the nonzero coefficients */
    int Arowndx[4] = { 1, 1, 1, 1};
    /* By default, all variables have a lower bound of zero
     * and an upper bound of infinity.  Therefore pass NULL
     * pointers in order to use these default values. */
    double *lb = NULL, *ub = NULL;
```

```
/*****************************************************************
 * Step 4: Specify and load the quadratic matrix
 *****************************************************************/
   /* The number of nonzeros in the quadratic matrix */
    int Qnz = 7;
    /* The nonzero coefficients in the Q-matrix */
    double Q[7] = { 1.00, .64, .27,
                          1.00, .13,
                                1.00,
                                      1.00} ;
    /* Specify the row indices of the nonzero coefficients in the
       Q-matrix. */
    int Qrowndx[7] = { 0, 0, 0, 0, 0, 0, 0};
    /* The indices of variables in the Q-matrix */
    int Qcolndx1[7] = {  0, 1, 2, 1, 2, 2, 3};
    int Qcolndx2[7] = {  0, 0, 0, 1, 1, 2, 3};
    /* Pass the linear portion of the data to problem structure
     * by a call to LSloadLPData() */
    nErrorCode = LSloadLPData( pModel, nM, nN, objsense, objconst,
                               reward, rhs, contype,
                               Anz, Abegcol, Alencol, A, Arowndx,
                               lb, ub);
    APIERRORCHECK;
   /* Pass the quadratic portion of the data to problem structure
    * by a call to LSloadQCData()  */
    nErrorCode = LSloadQCData(pModel, Qnz, Qrowndx,
                               Qcolndx1, Qcolndx2, Q );
    APIERRORCHECK;
 }

/*****************************************************************
 * Step 5: Perform the optimization using the barrier solver
 *****************************************************************/
 nErrorCode = LSoptimize( pModel, LS_METHOD_BARRIER,NULL);
 APIERRORCHECK;

/*****************************************************************
 * Step 6: Retrieve the solution
 *****************************************************************/
 {
    int i;
    double W[4], dObj;
 /* Get the value of the objective */
    nErrorCode = LSgetInfo( pModel, LS_DINFO_POBJ, &dObj) ;
    APIERRORCHECK;
    printf( "* Objective Value = %10g\n\n", dObj);
  /* Get the portfolio */
    nErrorCode = LSgetPrimalSolution ( pModel, W);
    APIERRORCHECK;
    printf ("* Optimal Portfolio : \n");
    for (i = 0; i < nN; i++)
    printf( "Invest %5.2f percent of total budget in asset %d.\n",
             100*W[i],i+1 );
    printf ("\n");
 }
```

```
  /*************************************************************
   * Step 7: Delete the LINDO environment
   *************************************************************/
   nErrorCode = LSdeleteEnv( &pEnv);
   /* Wait until user presses the Enter key */
   printf("Press <Enter> ...");
   getchar();
}
```

The source code file for this example may be found in the *\LINDOAPI\Samples\C\Markow* folder. After creating the executable "markow.exe", the application can be run from either with the *Start |Run* command, or from the DOS-prompt.

The output for the example follows:

```
* Objective Value =   0.173161
* Optimal Portfolio =
Invest 28.11 percent of total budget in asset 1.
Invest 21.78 percent of total budget in asset 2.
Invest  9.16 percent of total budget in asset 3.
Invest 40.96 percent of total budget in asset 4.
```

Example 2. Portfolio Selection with Restrictions on the Number of Assets Invested:

Another common portfolio selection model is the one where there is a restriction on the number of assets invested. This is also called the portfolio selection problem with cardinality constraints. In this formulation, it is also common to have bounds on the proportion of total assets invested to a particular asset type. The following example, given the required data, demonstrates how LINDO API is used to set up and solve such problems. Besides this example, the sample file *port.c* distributed with LINDO API can be used to solve the portfolio selection problems in J. E. Beasley's collection at ORLIB (http://mscmga.ms.ic.ac.uk/jeb/orlib/portinfo.html). You can find the source file in the *\LINDOAPI\SAMPLES\C\PORT* folder.

```
/* port.c

#######################################################################
#                       LINDO-API
#                    Sample Programs
#                  Copyright (c) 2007 by LINDO Systems, Inc
#
#         LINDO Systems, Inc.           312.988.7422
#         1415 North Dayton St.         info@lindo.com
#         Chicago, IL 60622             http://www.lindo.com
#######################################################################

  File   : port.c
  Purpose: Solve a quadratic mixed integer programming problem.
  Model  : Portfolio Selection Problem with a Restriction on
           the Number of Assets
```

```
            MINIMIZE   0.5 w'Q w
            s.t.   sum_i  w(i)                =  1
                   sum_i  r(i)w(i)           >=  R
                   for_i  w(i) - u(i) x(i) <=  0   i=1...n
                   sum_i  x(i)               <=  K
                   for_i  x(i) are binary          i=1...n
            where
            r(i)  : return on asset i.
            u(i)  : an upper bound on the proportion of total budget
                    that could be invested on asset i.
            Q(i,j): covariance between the returns of i^th and j^th
                    assets.
            K     : max number of assets allowed in the portfolio
            w(i)  : proportion of total budget invested on asset i
            x(i)  : a 0-1 indicator if asset i is invested on.

  Data:
  Covariance Matrix:
                 A1      A2      A3      A4      A5      A6      A7
       A1 [  1.00    0.11    0.04    0.02    0.08    0.03    0.10 ]
       A2 [  0.11    1.00    0.21    0.13    0.43    0.14    0.54 ]
       A3 [  0.04    0.21    1.00    0.05    0.16    0.05    0.20 ]
 Q = A4 [  0.02    0.13    0.05    1.00    0.10    0.03    0.12 ]
       A5 [  0.08    0.43    0.16    0.10    1.00    0.10    0.40 ]
       A6 [  0.03    0.14    0.05    0.03    0.10    1.00    0.12 ]
       A7 [  0.10    0.54    0.20    0.12    0.40    0.12    1.00 ]

  Returns Vector:
                 A1      A2      A3      A4      A5      A6      A7
   r =    [  0.14    0.77    0.28    0.17    0.56    0.18    0.70 ]

  Maximum Proportion of Total Budget to be Invested on Assets
                 A1      A2      A3      A4      A5      A6      A7
   u =    [  0.04    0.56    0.37    0.32    0.52    0.38    0.25 ]

  Target Return:
  R = 0.30

  Maximum Number of Assets:
  K = 3
*/
#include <stdlib.h>
#include <stdio.h>

/* LINDO API header file */
#include "lindo.h"

/* Define a macro to declare variables for
    error checking */
#define APIERRORSETUP
   int nErrorCode;
   char cErrorMessage[LS_MAX_ERROR_MESSAGE_LENGTH]
/* Define a macro to do our error checking */
#define APIERRORCHECK
   if (nErrorCode)
   {
```

```
        if ( pEnv)
        {
           LSgetErrorMessage( pEnv, nErrorCode,
            cErrorMessage);
           printf("Errorcode=%d:  %s\n", nErrorCode,
            cErrorMessage);
        } else {
           printf( "Fatal Error\n");
        }
        exit(1);
     }
  /* main entry point */
  int main()
  {
     APIERRORSETUP;
  /* Number of constraints */
     int nM = 10;
  /* Number of assets (7) plus number of indicator variables (7) */
     int nN = 14;
  /* declare an instance of the LINDO environment object */
     pLSenv pEnv = NULL;
  /* declare an instance of the LINDO model object */
     pLSmodel pModel;

     char MY_LICENSE_KEY[1024];
    /*************************************************************
     * Step 1: Create a LINDO environment.
     *************************************************************/
     nErrorCode = LSloadLicenseString(
                  "../../../license/lndapi60.lic",MY_LICENSE_KEY);
     APIERRORCHECK;

     pEnv = LScreateEnv ( &nErrorCode, MY_LICENSE_KEY);
     if ( nErrorCode == LSERR_NO_VALID_LICENSE)
     {
        printf( "Invalid License Key!\n");
        exit( 1);
     }
     APIERRORCHECK;

    /*************************************************************
     * Step 2: Create a model in the environment.
     *************************************************************/
     pModel = LScreateModel ( pEnv, &nErrorCode);
     APIERRORCHECK;
     {

    /*************************************************************
     * Step 3: Specify and load the LP portion of the model.
     *************************************************************/
       /* The maximum number of assets allowed in a portfolio */
        int  K = 3;
       /* The target return */
        double R = 0.30;
       /* The direction of optimization */
        int objsense = LS_MIN;
```

```
    /* The objective's constant term */
    double objconst = 0.;
    /* There are no linear components in the objective function.*/
    double c[14] = { 0., 0., 0., 0., 0., 0.,0.,
                     0., 0., 0., 0., 0., 0.,0.};
    /* The right-hand sides of the constraints */
    double rhs[10] = { 1.0, R, 0., 0., 0., 0., 0., 0., K};
    /* The constraint types */
    char contype[10] = {'E','G','L','L','L','L','L','L','L','L'};
    /* The number of nonzeros in the constraint matrix */
    int Anz = 35;
    /* The indices of the first nonzero in each column */
    int Abegcol[15] = { 0,  3,  6,  9, 12, 15, 18,
                       21, 23, 25, 27, 29, 31, 33,Anz};
    /* The length of each column. Since we aren't leaving
     * any blanks in our matrix, we can set this to NULL */
    int *Alencol = NULL;
    /* The nonzero coefficients */
    double A[35] = {  1.00, 0.14, 1.00,
                      1.00, 0.77, 1.00,
                      1.00, 0.28, 1.00,
                      1.00, 0.17, 1.00,
                      1.00, 0.56, 1.00,
                      1.00, 0.18, 1.00,
                      1.00, 0.70, 1.00,
                      -0.04, 1.00,
                      -0.56, 1.00,
                      -0.37, 1.00,
                      -0.32, 1.00,
                      -0.52, 1.00,
                      -0.38, 1.00,
                      -0.25, 1.00 };
    /* The row indices of the nonzero coefficients */
    int Arowndx[35] = { 0, 1, 2, 0, 1, 3, 0, 1, 4, 0, 1, 5,
                        0, 1, 6, 0, 1, 7, 0, 1, 8, 2, 9, 3,
                        9, 4, 9, 5, 9, 6, 9, 7, 9, 8, 9    };
    /* By default, all variables have a lower bound of zero
     * and an upper bound of infinity. Therefore pass NULL
     * pointers in order to use these default values. */
    double *lb = NULL, *ub = NULL;

/*************************************************************
 * Step 4: Specify and load the quadratic matrix
 *************************************************************/
 /* The number of nonzeros in the quadratic matrix */
    int Qnz = 28;
 /* The nonzero coefficients in the Q-matrix */
 double Q[28] = { 1.00,  0.11,  0.04,  0.02,  0.08,  0.03,  0.10,
                  1.00,  0.21,  0.13,  0.43,  0.14,  0.54,
                  1.00,  0.05,  0.16,  0.05,  0.20,
                  1.00,  0.10,  0.03,  0.12,
                  1.00,  0.10,  0.40,
                  1.00,  0.12,
                  1.00 };
```

```
    /* The row indices of the nonzero coefficients in the Q-matrix*/
    int  Qrowndx[28] = { -1, -1, -1, -1, -1, -1, -1,
                         -1, -1, -1, -1, -1, -1,
                         -1, -1, -1, -1, -1,
                         -1, -1, -1, -1,
                         -1, -1, -1,
                         -1, -1,
                         -1 };
    /* The indices of the first nonzero in each column in the Q-
  matrix */
    int Qcolndx1[28] = {  0, 1, 2, 3, 4, 5, 6,
                          1, 2, 3, 4, 5, 6,
                          2, 3, 4, 5, 6,
                          3, 4, 5, 6,
                          4, 5, 6,
                          5, 6,
                          6};
    int Qcolndx2[28] = {  0, 0, 0, 0, 0, 0, 0,
                          1, 1, 1, 1, 1, 1,
                          2, 2, 2, 2, 2,
                          3, 3, 3, 3,
                          4, 4, 4,
                          5, 5,
                          6};
    /* Pass the linear portion of the data to problem structure
     * by a call to LSloadLPData() */
     nErrorCode = LSloadLPData( pModel, nM, nN, objsense, objconst,
                                c, rhs, contype,
                                Anz, Abegcol, Alencol, A, Arowndx,
                                lb, ub);
     APIERRORCHECK;
    /* Pass the quadratic portion of the data to problem structure
     * by a call to LSloadQCData()  */
     nErrorCode = LSloadQCData(pModel, Qnz, Qrowndx,
                               Qcolndx1, Qcolndx2, Q);
     APIERRORCHECK;
    /* Pass the integrality restriction to problem structure
     * by a call to LSloadVarData()  */
     {
       char vartype[14] ={ 'C','C','C','C','C','C','C',   /* w(j) */
                           'B','B','B','B','B','B','B' }; /* x(j) */
       nErrorCode = LSloadVarType(pModel, vartype);
       APIERRORCHECK;
     }
   }
```

```
  /*********************************************************************
   * Step 5: Perform the optimization using the MIP solver
   *********************************************************************/
   nErrorCode = LSsolveMIP( pModel, NULL);
   APIERRORCHECK;
   {
  /*********************************************************************
   * Step 6: Retrieve the solution
   *********************************************************************/
      int i;
      double x[14], MipObj;
      /* Get the value of the objective and solution */
      nErrorCode = LSgetInfo(pModel, LS_DINFO_MIP_OBJ, &MipObj);
      APIERRORCHECK;

      LSgetMIPPrimalSolution( pModel, x) ;
      APIERRORCHECK;
      printf ("*** Optimal Portfolio Objective = %f\n", MipObj);
      for (i = 0; i < nN/2; i++)
        printf( "Invest %5.2f percent of total budget in asset
%d.\n",
                100*x[i],i+1 );
      printf ("\n");
   }
  /*********************************************************************
   * Step 7: Delete the LINDO environment
   *********************************************************************/
   nErrorCode = LSdeleteEnv( &pEnv);
 /* Wait until user presses the Enter key */
   printf("Press <Enter> ...");
   getchar();
}
```

After building this application, it can be run from the DOS-prompt to produce the following summary report on your screen.

```
*** Optimal Portfolio Objective = 0.192365
Invest  0.00 percent of total budget in asset 1.
Invest  0.00 percent of total budget in asset 2.
Invest  0.00 percent of total budget in asset 3.
Invest 32.00 percent of total budget in asset 4.
Invest 32.76 percent of total budget in asset 5.
Invest 35.24 percent of total budget in asset 6.
Invest  0.00 percent of total budget in asset 7.
```

Chapter 6: Solving Second-Order Cone Programs

The optimization capabilities of LINDO API extend to the solution of second-order-cone problems (SOCP) of the following form

Optimize $||A_0 x + b_0|| + c_0 x$
subject to:
$||A_i x + b_i|| + c_i x - d_i \; ? \; 0$ for $i = 0,1,\ldots,m-1$,
$L_j \le x_j \le U_j$ for $j = 0,1,\ldots,n-1$,
x_j is integer for j in a specified $J \subseteq \{0,\ldots, n-1\}$

where
Optimize is either minimize or maximize,
A_i are matrices of appropriate dimensions $i=0,\ldots,m-1$,
b_i and c_i are vectors of constants,
d_i are constants,
$x = \{x_0, x_2,\ldots,x_{n-1}\}$, is an n-vector of decision variables.
"?" is one of the relational operators "≤", "=", or "≥".

Without the integrality restrictions, SOCPs are nonlinear convex problems that include linear and convex quadratically constrained quadratic programs as special cases. Several decision problems in engineering design and control can be formulated as SOCP. LINDO API solves this class of problems using the so-called *conic optimizer*, which uses an interior-point algorithm. To solve a convex problem using LINDO API, it may be advantageous to cast the problem (e.g. a QCQP) as a SOCP and use the conic optimizer. It has been empirically observed that the conic optimizer is generally faster than the default barrier solver.

To motivate the second-order cone problems and common forms of quadratic cones, consider the following two constraints:

$x^2 + y^2 - z^2 \le 0$,
$z \ge 0$

Geometrically, the feasible region defined by these two constraints is an *ice cream cone*, with the point of the cone at **(0,0,0)**. The feasible region for the constraint $x^2 + y^2 - z^2 \leq 0$ by itself is not convex. The feasible region consists of two ice cream cones, one right side up, the other upside down, and with their pointy ends touching. The constraint $z \geq 0$ eliminates the upside down cone and leaves the *quadratic cone* illustrated in Figure 5. Second-order cone problems are essentially a generalization of linear models defined over polyhedral cones to ones defined over quadratic cones.

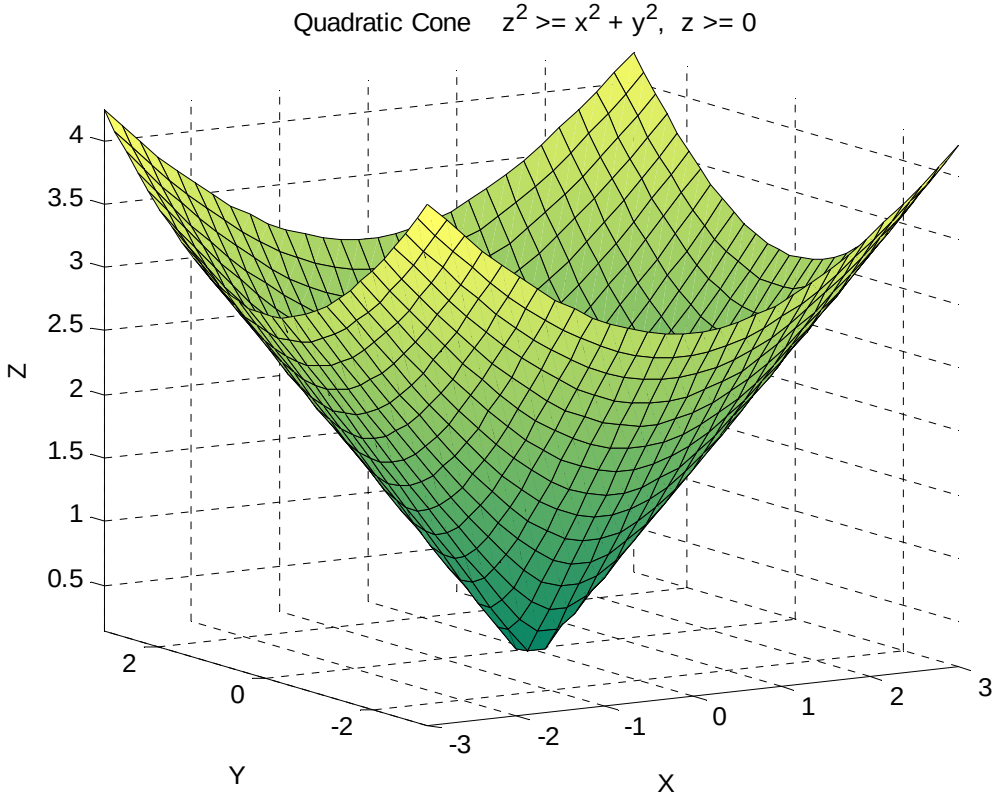


Figure 5. Quadratic Cone

More generally, in n dimensions, a simple quadratic cone (ice-cream cone) constraint is of the form:

$$-x_0^2 + x_1^2 + x_2^2 + \ldots + x_n^2 \leq 0;$$
$$x_0 \geq 0;$$

Second-order cone constraints are more general than they might at first appear. For another conic form, consider the constraints:

$$-uv + x^2 \leq 0,$$
$$u, v \geq 0.$$

The first constraint by itself describes a nonconvex feasible region (colored blue and green) illustrated in Figure 6. The three constraints together, however, describe a convex feasible region (colored green only) called the *rotated quadratic cone*.

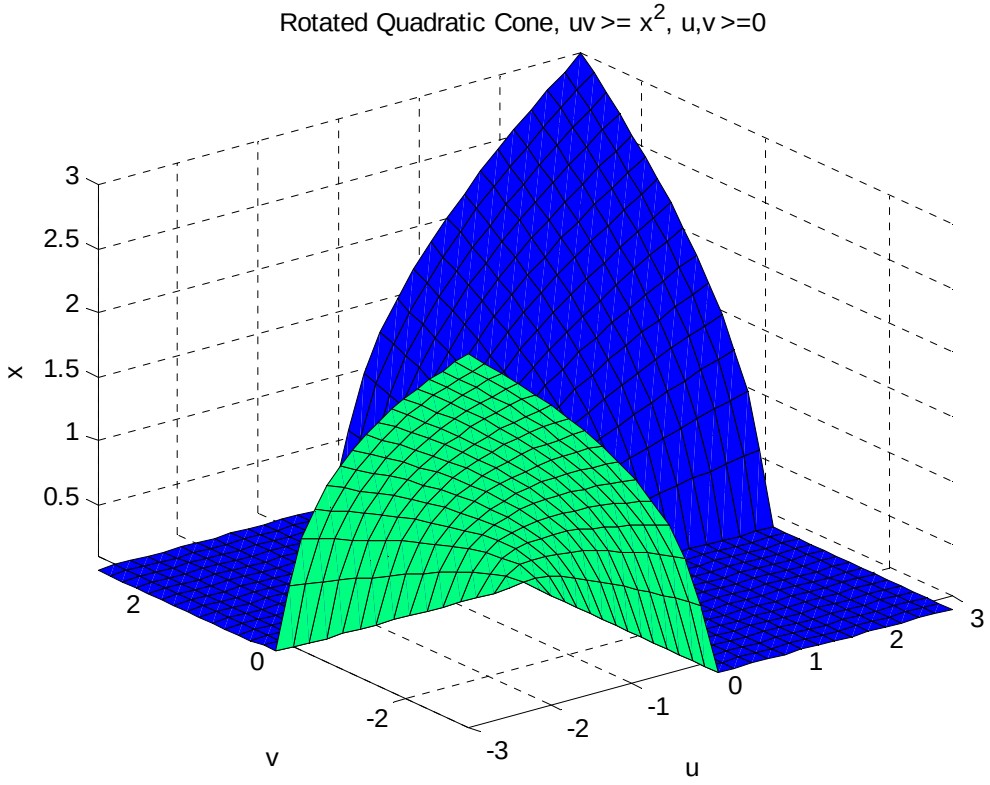


Figure 6. Rotated Quadratic Cone

More generally, in n dimensions, the rotated quadratic cone constraint in standard form is:

$$-2x_0 x_1 + x_2^2 + x_3^2 + \ldots + x_n^2 \leq 0;$$
$$x_0, x_1 \geq 0;$$

In both simple and rotated quadratic cones, a variable can appear in at most one cone constraint. If naturally you would like to have a variable, say x_2, appear in two cone constraints, then you must introduce an extra copy of the variable, say y_2 , for the second cone constraint and then connect the two with the linear constraint $x_2 - y_2 = 0$.

Notice, using a standard transformation, rotated quadratic cone constraints can be shown to be equivalent to quadratic cone constraints:

$$y = (u - v)/2,$$
$$z = (u + v)/2,$$
$$x^2 + y^2 - z^2 \leq 0,$$
$$z \geq 0.$$

Setting up Second-Order Cone Programs

There are two ways by which you can setup a second-order-cone program with LINDO API. The first one is reading the model directly from an MPS file, which uses an extended format to incorporate quadratic cones in model formulation. The second way is to build the model directly in memory and pass the data representing quadratic cones to LINDO API. Due to their special structure, LINDO API does not require cone constraints to be included in model body explicitly. In either approach, the following data suffice to characterize the cone constraints:

- type of each cone (quadratic or rotated-quadratic),
- ordered set of variables characterizing each cone.

Loading Cones via Extended MPS Format Files

The cone constraints can be described in an MPS file by adding a *CSECTION* section to the file for each cone. The format of the *CSECTION* section is simple. It contains four tokens per *CSECTION* line, and the names of variables that comprise the cone in the following lines. The second token in the *CSECTION* line stands for the name of the cone. The third token is reserved and arbitrarily set to 0. The fourth token stands for cone type (*QUAD* or *RQUAD*). The token *QUAD* stands for quadratic cones (e.g. Figure 5) and the token *RQUAD* stands for rotated-quadratic cones (e.g. Figure 6). Each *CSECTION* line is followed by the names of variables (one per line) describing the cone. The ordering of variables is not important except for the first two. For *QUAD* cones, the first variable in the list should be the variable that stands for the depth of the cone, e.g. variable *z* in Figure 5. For *RQUAD*, the first two variables in the list should be the ones that form the product of two variables (e.g. variables u and v in Figure 6).

Consider the following second-order cone model. The single cone constraint in the model appears after constraint 2, without which the model is a simple linear model.

Minimize			w			
s.t.						
Constraint 0:			$19\,x_1 + 21\,x_3 + 21\,x_4$		=	1
Constraint 1:			$12\,x_1 + 21\,x_2$		=	1
Constraint 2:			$12\,x_2 + 16\,x_5$		=	1
QUAD Cone:			$-w + (x_1^2 + \ldots + x_5^2)^{0.5}$		≤	0
	$-\infty$	≤	x_i		≤	$+\infty$
	0	≤	w		≤	$+\infty$

The cone constraint is a simple *quadratic* cone defined over variables $\{w, x_1, x_2, x_3, x_4, x_5\}$. This reflects to the MPS file in the following fashion.

```
NAME           MININORM
ROWS
 N       OBJ
 E  R0000000
 E  R0000001
 E  R0000002
 L  R0000003
COLUMNS
    W              OBJ                1
    X0000001  R0000000               19
    X0000001  R0000001               12
    X0000002  R0000001               21
    X0000002  R0000002               12
    X0000003  R0000000              -17
    X0000004  R0000000               21
    X0000005  R0000002               16
RHS
    RHS1      R0000000                1
    RHS1      R0000001                1
    RHS1      R0000002                1
BOUNDS
 FR BND1      X0000001
 FR BND1      X0000002
 FR BND1      X0000003
 FR BND1      X0000004
 FR BND1      X0000005
CSECTION      CONE0000                  0  QUAD
* The first variable in this section must be the 'x0' variable
    W
    X0000001
    X0000002
    X0000003
    X0000004
    X0000005
ENDATA
```

Note: Your license must have the barrier or nonlinear license options to be able to work with second-order cone formulations. Attempting to solve a problem that has cone data using other optimization algorithms such as primal simplex, dual simplex, or mixed-integer solver will return an error.

Loading Cones via API Functions

The second way to input cone data is by setting-up a problem structure and using LINDO API's cone programming functions to specify the cone constraints. In this framework, your front-end program should perform at least the following steps to enter the problem and retrieve its solution:

- Create a LINDO environment with a call to *LScreateEnv*().
- Create a model structure in this environment with a call to *LScreateModel*().
- Load problem structure and linear data into the model structure with a call to *LSloadLPData*().
- Load the cone data into the model structure with a call to *LSloadConeData*().
- Load (optionally) the integer-programming data with a call to *LSloadVarType*().
- Solve the problem with a call to *LSoptimize*() (or *LSsolveMIP*() if there are integer variables).
- Retrieve the solution with calls to *LSgetInfo*(), *LSgetPrimalSolution*(), and *LSgetDualSolution*().
- Delete the model and environment with a call to *LSdeleteEnv*().

The step specific to loading cone data is Step 4 where cone types and variable indices are loaded to the model. Assuming the model has *nCone* cones, and a total of *nNz* variables in all cones, the following three-vector representation is sufficient to store all necessary data.

```
char acConeTypes[nCone] = {'Q', 'Q', … , 'R' , 'R' };
int  anConeStart[nCone + 1] = {0, k1, k2, … , knCone-1, nNz};
int  anConeVars[nNz] = {j1,…,jk1,…,jk2,…,jk(nCone-1)}
```

Notice, *anConeStart*[k_c] marks the beginning position in vector *anConeVars*[] keeping variables in cone *c*. This convention is similar to the one used in the representation of coefficient matrices in LPs. In the following, the complete source code for setting up the example above is given.

Example 3: Minimization of Norms:

One of the common types of second-order-cone problems is the minimization of norms. This problem has applications in several areas such as optimal location problems, statistics and engineering design. This problem has the following general form.

$$\text{Minimize } \Sigma\, z^{(j)}$$
$$Dx = b$$
$$\| A^{(j)}x + b^{(j)}\| \le z^{(j)} \qquad \text{for all } j = 1,\ldots,p$$

where

$z^{(j)}$ is a scalar decision variable for all $j = 1,\ldots,p$,
$x = \{x_1, x_2,\ldots,x_n\}$ is a vector of decision variables.
D is an m by $n$matrix
b is a m vector
$A^{(j)}$ is an n_j by n matrix of constants, for j=1,…,p,
$b^{(j)}$ is a 1 by n_j vector of constants, for j=1,…,p,

The following sample code shows how to set up and solve a norm minimization problem using LINDO API's conic solver.

```
/*
#################################################################
#                        LINDO-API
#                      Sample Programs
#                   Copyright (c) 2007 by LINDO Systems, Inc
#
#         LINDO Systems, Inc.           312.988.7422
#         1415 North Dayton St.         info@lindo.com
#         Chicago, IL 60622             http://www.lindo.com
#################################################################

   File   : ex_soc1.c
   Purpose: Solve a second-order cone program.
   Model  : Simple norm minimization

            MINIMIZE      w
            subject to            A.x    >= b
                         -w^2  + ||x||^2 <= 0
            x  : an n-vector
            w  : the norm of vector x.
   Data:
   A-matrix for linear constraints:
              w     x1    x2    x3    x4   x5
         [  0     19     0    21    21    0   ]
     A = [  0     12    21    0     0     0   ]
         [  0     0     12    0     0    16   ]

   b-vector:
     b =  [  1     1     1 ];

*/
#include <stdlib.h>
#include <stdio.h>
#include "lindo.h"

/* Define a macro to declare variables for error checking */
#define APIERRORSETUP
    int nErrorCode;
    char cErrorMessage[LS_MAX_ERROR_MESSAGE_LENGTH]

/* Define a macro to do our error checking */
#define APIERRORCHECK
    if (nErrorCode)
    {
       if ( pEnv)
       {
          LSgetErrorMessage( pEnv, nErrorCode,
           cErrorMessage);
          printf("Errorcode=%d:  %s\n", nErrorCode,
           cErrorMessage);
       } else {
          printf( "Fatal Error\n");
       }
       exit(1);
    }
/* main entry point */
```

```
int main()
{
    int nSolStatus;

    APIERRORSETUP;

    int nM = 4; /* Number of constraints */

    int nN = 6; /* Number of variables */

    pLSenv pEnv;

    pLSmodel pModel;

    char MY_LICENSE_KEY[1024];
   /*********************************************************
    * Step 1: Create a model in the environment.
    *********************************************************/
    nErrorCode = LSloadLicenseString(
      "../../../license/lndapi60.lic",MY_LICENSE_KEY);
    pEnv = LScreateEnv ( &nErrorCode, MY_LICENSE_KEY);
    if ( nErrorCode == LSERR_NO_VALID_LICENSE)
    {
       printf( "Invalid License Key!\n");
       exit( 1);
    }
    APIERRORCHECK;

   /*********************************************************
    * Step 2: Create a model in the environment.
    *********************************************************/
    pModel = LScreateModel ( pEnv, &nErrorCode);
    APIERRORCHECK;

    {
   /*********************************************************
    * Step 3: Specify the linear portion of the model.
    *********************************************************/

      /* The direction of optimization */
       int objsense = LS_MIN;

       /* The objective's constant term */
       double objconst = 0.;

       /* The coefficients of the objective function*/
       double cost[6] = { 1., 0., 0., 0., 0., 0.};

       /* The right-hand sides of the constraints */
       double rhs[4] = { 1.0, 1.0, 1.0,  0.0 };

       /* The constraint types */
       char contype[4] = {'E','E','E','L'};

       /* The number of nonzeros in the constraint matrix */
       int Anz = 7;
```

```
    /* The indices of the first nonzero in each column */
    int Abegcol[7] = { 0, 0, 2, 4, 5, 6, Anz};

    /* The length of each column.  Since we aren't leaving
     * any blanks in our matrix, we can set this to NULL */
    int *Alencol = NULL;

    /* The nonzero coefficients */
    double A[7] = { 19, 12 , 21, 12, -17, 21, 16};

    /* The row indices of the nonzero coefficients */
    int Arowndx[7] = { 0, 1, 1, 2, 0, 0, 2};

    /* All variables are free */
    double lb[6] = { 0.000000000,-LS_INFINITY,-LS_INFINITY,
                    -LS_INFINITY,-LS_INFINITY,-LS_INFINITY};

    double ub[6] = {LS_INFINITY,LS_INFINITY,LS_INFINITY,
                    LS_INFINITY,LS_INFINITY,LS_INFINITY};
/*****************************************************
 **Step 4: Specify the QCONE data
 *****************************************************/

 /** The number of CONE constraints*/
    int nCones = 1;

 /** Specify the column indices of variables in the CONE
     constraint,*/
    int paiConecols[6] = {  0, 1, 2, 3, 4, 5};

    int paiConebeg[2] = {0, 6};

 /** Specify cone type */
    char pszConeTypes[1] = { LS_CONETYPE_QUAD };

   /* Pass the linear portion of the data to problem structure
    * by a call to LSloadLPData() */

    nErrorCode = LSloadLPData( pModel, nM, nN, objsense, objconst,
                               cost, rhs, contype,
                               Anz, Abegcol, Alencol, A, Arowndx,
                               lb, ub);
    APIERRORCHECK;

   /* Pass the cone portion of the data to problem structure
    * by a call to LSloadConeDataData()  */
    nErrorCode = LSloadConeData(pModel, nCones, pszConeTypes,
      paiConebeg, paiConecols);
    APIERRORCHECK;

 /** Export the conic model in case required */
    LSwriteMPSFile(pModel,"cone.mps",0);

 }
/*****************************************************
```

```
     * Step 5: Perform the optimization using the QCONE solver
    ******************************************************/
     nErrorCode = LSsetModelIntParameter(pModel,
      LS_IPARAM_BARRIER_SOLVER, LS_BAR_METHOD_FREE);

     nErrorCode = LSoptimize( pModel, LS_METHOD_FREE, &nSolStatus);
     APIERRORCHECK;
    /******************************************************
     * Step 6: Retrieve the solution
     ******************************************************/
     if (nSolStatus == LS_STATUS_OPTIMAL ||
         nSolStatus == LS_STATUS_BASIC_OPTIMAL)
     {
        int i;
        double x[6], dObj;
        /* Get the value of the objective */
        nErrorCode = LSgetInfo( pModel, LS_DINFO_POBJ, &dObj) ;
        APIERRORCHECK;

        nErrorCode = LSgetPrimalSolution ( pModel, x);
        APIERRORCHECK;

        printf("Minimum norm = %11.5f*\n",x[0]);
        for (i = 0; i < nN; i++)
          printf("%7s x[%d] = %11.5f\n","",i,x[i] );
        printf ("\n");
     }
     else
     {
       printf("Not optimal, status = %d\n",nSolStatus);
     }

    /******************************************************
     * Step 7: Delete the LINDO environment
     ******************************************************/
     nErrorCode = LSdeleteEnv( &pEnv);

    /* Wait until user presses the Enter key */
     printf("Press <Enter> ...");
     getchar();

}
```

The source code file for this example may be found in the \LINDOAPI\samples\c\ex_soc1\ folder. After creating the executable "ex_soc1.exe", you can run the application from the DOS-prompt and produce the following report on your screen.

```
Minimum norm =     0.05903
        x[0] =     0.05903
        x[1] =     0.02083
        x[2] =     0.03572
        x[3] =    -0.01407
        x[4] =     0.01738
        x[5] =     0.03
```

Converting Models to SOCP Form

SOCP constraints are more general than perhaps is superficially obvious. As a first illustration of this generality, consider a constraint set of the form:

$r \geq (a + bx)/(c+dx)$;
$c+dx \geq 0$;

Expressions such as this arise for example in modeling delay as a function of traffic volume through a congested facility or transportation link. A constraint such as the above can be put into SOCP form if $a - bc/d \geq 0$. To do this define:

$2y = c+dx$, then $x = (2y-c)/d$, and $r \geq (a + bx)/(c+dx) = (a + bx)/(2y) = (a - bc/d)/(2y) + b/d$.

Thus, the constraint is convex if $y \geq 0$ and $a - bc/d \geq 0$.

If we define $u = (r-b/d)$, then $r - b/d \geq (a - bc/d)/(2y)$ is equivalent to the cone constraint:

$2yu \geq a-bc/d$.

Summarizing, given $a - bc/d \geq 0$, we can replace:

$r \geq (a + bx)/(c+dx)$;
$c+dx \geq 0$;

by the SOCP set of constraints:

$2y = c+dx$;
$r = u + b/d$;
$2yu \geq a-bc/d$;
$y \geq 0$;

The follow code shows how use LINDO API's conic solver to set up and solve a model with constraints of the above type, where $b = c = 0$.

Example 4: Ratios as SOCP Constraints:

```
/*
#################################################################
#                        LINDO-API
#                     Sample Programs
#                  Copyright (c) 2007 by LINDO Systems, Inc
#
#         LINDO Systems, Inc.           312.988.7422
#         1415 North Dayton St.         info@lindo.com
#         Chicago, IL 60622             http://www.lindo.com
#################################################################

  File   : ex_soc2.c
  Purpose: Solve a second-order rotated cone program.
          A rotated cone constraint is of the form:
            2*x0*x1 - x2*x2 - x3*x3 - ... >= 0;
              x0, x1 >= 0;

    The model in natural form:
           MINIMIZE      11*x0 +  7*x1  + 9*x2;
           subject to     5/x0 +  6/x1  + 8/x2 <= 1;
                          x0, x1, x2 >= 0;

       Reformulated as a rotated cone:
           MINIMIZE      11*x0 +  7*x1  + 9*x2;
           subject to     2*r0 +  2*r1  + 2*r2 <= 1;
                            k0                 = 5^0.5;
                                    k1         = 6^0.5;
                                            k2 = 8^0.5
                       2*r0*x0>= k0^2;
                       2*r1*x1>= k1^2;
                       2*r2*x2>= k2^2;
                          x0, x1, x2 >= 0;
                          r0, r1, r2 >= 0;

  The constraint matrix for the linear constraints:
             0   1   2   3   4   5   6   7   8
            x0  x1  x2  r0  r1  r2  k0  k1  k2
         [  0   0   0   2   2   2   0   0   0] <= 1
    A =  [  0   0   0   0   0   0   1   0   0] = 5^0.5
         [  0   0   0   0   0   0   0   1   0] = 6^0.5
         [  0   0   0   0   0   0   0   0   1] = 8^0.5
*/
#include <stdlib.h>
#include <stdio.h>
#include "lindo.h"

/* Define a macro to declare variables for error checking */
#define APIERRORSETUP
   int nErrorCode;
   char cErrorMessage[LS_MAX_ERROR_MESSAGE_LENGTH]

/* Define a macro to do our error checking */
```

```
#define APIERRORCHECK
   if (nErrorCode)
   {
      if ( pEnv)
      {
         LSgetErrorMessage( pEnv, nErrorCode,
          cErrorMessage);
         printf("Errorcode=%d:  %s\n", nErrorCode,
          cErrorMessage);
      } else {
         printf( "Fatal Error\n");
      }
      exit(1);
   }

/* main entry point */
int main()
{
   int nSolStatus;

   APIERRORSETUP;

   int nM = 4; /* Number of linear constraints */

   int nN = 9; /* Number of variables */

   pLSenv pEnv;

   pLSmodel pModel;

   char MY_LICENSE_KEY[1024];

  /*********************************************************
   * Step 1: Create a model in the environment.
   *********************************************************/
  // Load the license into MY_LICENSE_KEY
   nErrorCode = LSloadLicenseString( "../../../lndapi60.lic",
   MY_LICENSE_KEY);
   pEnv = LScreateEnv ( &nErrorCode, MY_LICENSE_KEY);
   if ( nErrorCode == LSERR_NO_VALID_LICENSE)
   {
      printf( "Invalid License Key!\n");
      exit( 1);
   }
   APIERRORCHECK;

  /*********************************************************
   * Step 2: Create a model in the environment.
   *********************************************************/
   pModel = LScreateModel ( pEnv, &nErrorCode);
   APIERRORCHECK;
   {
  /*********************************************************
   * Step 3: Specify the linear portion of the model.
   *********************************************************/
     /* The direction of optimization */
```

```
    int objsense = LS_MIN;

    /* The objective's constant term */
    double objconst = 0.;

    /* The coefficients of the objective function*/
    double cost[9] = { 11.0, 7.0, 9.0 , 0., 0., 0., 0., 0., 0.};

   /* The right-hand sides of the constraints( square roots of 5,
   6, 8)*/
    double rhs[4] = { 1.0, 2.2360679775, 2.44948974278,
   2.82842712475};

    /* The constraint types */
    char contype[4] = {'L', 'E', 'E', 'E'};

    /* The number of nonzeros in the constraint matrix */
    int Anz = 6;

    /* The indices in A[] of the first nonzero in each column */
    int Abegcol[10] = { 0, 0, 0, 0, 1, 2, 3, 4, 5, Anz};

    /* The length of each column.  Since we aren't leaving
     * any blanks in our matrix, we can set this to NULL */
    int *Alencol = NULL;

    /* The nonzero constraint coefficients */
    double A[6] = { 2.0, 2.0, 2.0, 1.0, 1.0, 1.0};

    /* The row indices of the nonzero coefficients */
    int Arowndx[6] = { 0, 0, 0, 1, 2, 3};

    /* All variables are non-negative */
    double lb[9] = { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

    double ub[9] = {LS_INFINITY,LS_INFINITY,LS_INFINITY,
                    LS_INFINITY,LS_INFINITY,LS_INFINITY,
                    LS_INFINITY,LS_INFINITY,LS_INFINITY};
/*********************************************************
 * Step 4: Set up data describing the CONE constraints
 *********************************************************/
   /* The number of Cone constraints */
    int nCones = 3;

   /* The col indices of the variables in each Cone constraint */
    int paiConecols[9] = {0, 3, 6,  1, 4, 7,  2, 5, 8};

   /* The start in paiConecols[] of the indices for each Cone
    constraint */
    int paiConebeg[4] =  {0,        3,        6,      9};

   /* These are Rotated Cone constraints */
    char pszConeTypes[3] = { 'R', 'R', 'R'};

   /* Pass the linear portion of the data to problem structure
    * by a call to LSloadLPData() */
```

```
      nErrorCode = LSloadLPData( pModel, nM, nN, objsense, objconst,
                                 cost, rhs, contype,
                                 Anz, Abegcol, Alencol, A, Arowndx,
                                 lb, ub);
      APIERRORCHECK;
     /* Pass the Cone portion of the data to the problem structure
      * by a call to LSloadConeData()  */
      nErrorCode = LSloadConeData(pModel, nCones, pszConeTypes,
        paiConebeg, paiConecols);
      APIERRORCHECK;

/* Optionally, write an MPS file version of the model */
      LSwriteMPSFile(pModel,"cone.mps",0);
   }
  /*********************************************************
   * Step 5: Perform the optimization using the QCONE solver
   *********************************************************/
   nErrorCode = LSsetModelIntParameter(pModel,
  LS_IPARAM_BARRIER_SOLVER, LS_BAR_METHOD_FREE);

   nErrorCode = LSoptimize( pModel, LS_METHOD_FREE, &nSolStatus);
   APIERRORCHECK;
  /*********************************************************
   * Step 6: Retrieve the solution
   *********************************************************/
   if (nSolStatus == LS_STATUS_OPTIMAL || nSolStatus ==
    LS_STATUS_BASIC_OPTIMAL)
   {
      int i;
      double x[9], dObj;
      /* Get the value of the objective */
      nErrorCode = LSgetInfo( pModel, LS_DINFO_POBJ, &dObj) ;
      APIERRORCHECK;

      nErrorCode = LSgetPrimalSolution ( pModel, x);
      APIERRORCHECK;
      printf("        Obj =  %11.5f\n",dObj);
      for (i = 0; i < nN; i++)
        printf("%7s x[%d] = %11.5f\n","",i,x[i] );
      printf ("\n");
   }
   else
   {
     printf("Not optimal, status = %d\n",nSolStatus);
   }

  /*********************************************************
   * Step 7: Delete the LINDO environment
   *********************************************************/
   nErrorCode = LSdeleteEnv( &pEnv);

  /* Wait until user presses the Enter key */
   printf("Press <Enter> ...");
   getchar();
}
```

The source code file for this example may be found in the \LINDOAPI\samples\c\ex_soc2\ folder.

After creating the executable "ex_soc2.exe", you can run the application from the DOS-prompt and produce the following report on your screen.

```
Obj =    500.96379
x[0] =    15.09022
x[1] =    20.72079
x[2] =    21.10287
x[3] =     0.16567
x[4] =     0.14478
x[5] =     0.18955
x[6] =     2.23607
x[7] =     2.44949
x[8] =     2.82843

Press <Enter> ..
```

Quadratic Programs as SOCP

Although perhaps not immediately obvious, a SOCP is at least as general as a quadratic program. In a quadratic program one typically wants to either minimize a quadratic expression, written as $x'Qx$, or constrain $x'Qx$ from above. A related example is in Value-At-Risk analysis, where one may have models of the form:

Minimize $k*\sigma - \mu$;
Subject to
$\sigma^2 \geq x'Qx$;
$\mu = r'x$;

If the Q matrix is positive definite, then $x'Qx$ is convex and SOCP can be applied as outlined below. An easy way to a) check for positive definiteness, and b) put the model into a SOCP form is to compute a Cholesky Decomposition or "square root" of the Q matrix. In matrix notation we can write:

$$\sigma^2 \geq x\,Q\,x' = x\,L\,L'\,x'.$$

Here, L is an lower triangular matrix which we can think of as the square root of Q. The following LINGO code will compute L:

```
!Compute the Cholesky factor L, so LL'= Q;
@FOR( ASSET( I):
  @FOR( MAT( I, J)| J #LT# I:
    ! Should watch for divide by zero here...;
     L(I,J)= ( Q( I, J) - @SUM( MAT( I, K)| K #LT# J:
                L( I, K) * L( J, K)))/ L( J, J);
              );
     L(I,I)= ( Q( I, I) - @SUM( MAT( I, K)| K #LT# I:
               L( I, K) * L( I, K)))^.5;
             );
```

A key observation is that Q is strictly positive definite if and only if $L(i,i) > 0$ at every step of the above computation. Thus, given a quadratic expression, we can try to compute the Cholesky decomposition of its coefficients. If we succeed, then we can replace

$$\sigma^2 \geq x'Q\,x = x'L'Lx.$$

by the cone constraints:

$w = x\,L,$
$\sigma^2 \geq w\,w';$

As an example, suppose we wish to use the following covariance matrix:

```
       0.01080753  0.01240721  0.01307512
Q =    0.01240721  0.05839169  0.05542639
       0.01307512  0.05542639  0.09422681
```

The Cholesky factorization of Q is:

```
       0.10395930
L =    0.11934681  0.21011433
       0.1257716   0.19235219  0.20349188
```

Notice that 0.10395930^2 = 0.01080753.

We can replace the expression involving 9 quadratic terms (more accurately, 6):

```
σ^2    ≥ 0.01080753*x1*x1 + 0.01240721*x1*x1 + 0.01307512*x1*x3
       + 0.01240721*x2*x1 + 0.05839169*x2*x2 + 0.05542639*x2*x3
       + 0.01307512*x3*x1 + 0.05542639*x3*x2 + 0.09422681*x3*x3;
```

by three linear expressions and one nonlinear expression involving 3 quadratic terms:

```
w1    = 0.10395930*x1;
w2    = 0.11934681*x1 + 0.21011433*x2;
w3    = 0.1257716 *x1 + 0.19235219*x2 + 0.20349188*x3;
σ^2   ≥ w1*w1 + w2*w2 + w3*w3;
```

which is a SOCP type constraint

Chapter 7: Solving Nonlinear Programs

The nonlinear programming (NLP) interface of LINDO API is designed to solve optimization problems of the form:

optimize $f(\mathbf{x})$;
such that

$g_i(\mathbf{x})$? b_i	for $i = 0$ to m-1:
$L_j \le x_j \le U_j$	for $j = 0$ to $j = n - 1$:
x_j is integer	for j in a specified $J \subseteq \{0, \ldots, n\text{-}1\}$

where

optimize is either minimize or maximize,
$f(x)$ and $g_i(x)$ are scalar valued real functions.
$\mathbf{x}$ is the vector of all variables, and
"?" is one of the relational operators "≤", "=", or "≥".

For nonlinear constraints, b_i is assumed to be zero.

Nonlinear programs are the most general form of mathematical models solvable by LINDO API. They contain all other types of models that have been covered so far as special cases. It is useful to think of a mathematical modeling system as consisting of at least three layers: 1) a frontend/user interface that accepts a model in user-convenient form and converts it to solver-convenient form, 2) a solver manager that looks at the model in solver form and decides how to solve the model, and 3) solvers that do the low level solving (e.g., a primal simplex solver, barrier solver, network solver, general nonlinear solver, etc.).

LINDO API performs levels (2) and (3) and provides tools for setting up and passing a model from level (1) to level (2). As we have seen, linear or quadratic (mixed-integer) models can be fully described by (sparse) matrices and vectors. Thus, setting up a model in one of these types involves the simple tasks of: (i) creating the sparse representation of the constraint matrix; (ii) building the objective function, right-hand-side, variable-bound, and variable-type vectors along with some others; and (iii) passing these data objects to LINDO API for solution (levels 2 and 3).

The description of nonlinear models, however, is not as easy as for linear or quadratic (mixed-integer) models. The main issue lies in the evaluation of the functional values of the objective function and the constraints with respect to a solution vector. This is because the constraints and objective function are not linear and hence cannot be represented with matrices and vectors alone. However, if one has the means for evaluating nonlinear expressions at a given solution, then sparse matrix representation technique becomes an efficient tool for the nonlinear solver to manipulate working-matrices that depend on the nonzero structure of the constraint matrix and their gradients.

The LINDO API offers two basic interface styles for describing NLP's: a) "black-box" style, and b) "instruction-list" style, plus a combination of the two called c)"grey-box" style. In the black-box style, the model developer achieves this by means of an external function (e.g., written in C or C++) that evaluates the nonlinear expressions at a given solution, and then passes the name of this routine to LINDO API. Under the instruction-list style, the user passes a set of instruction lists to LINDO API. These instruction-lists describe how to compute the value of each row of the model. It is possible for the instruction-list interface to inherit the black-box interface via a special instruction operator. This feature lends itself to a combination of these two basic interfaces, called the grey-box interface.

There are certain advantages to each of these approaches. If a standard programming language is used by the black-box interface, the computation of the value of a row is very efficient. In the instruction-list interface, LINDO API is given more detailed information about each row of the model, rather than having this information hidden in a black box. For example, if a row of the model is in fact linear, then LINDO API will detect this and exploit it. The following sections, describe in detail how each style could be used in setting up a mathematical programming model as general as nonlinear (integer) models.

Black-Box Style Interface

One of the critical issues in efficient solution of NLP models using LINDO API's black-box style interface is the exploitation of linearity and sparsity. The NLP solver can exploit linearity in the model where it exists if the user (optionally) points out in advance the location of the linear and nonlinear elements. This also allows the solver to partially see "inside the black box" to the extent that the user provides information about the nonzero structures of I) the linear terms (step 3 below), and II) the derivatives of nonlinear terms in the model (step 4 below).

The other issue specific to black-box interface is the set-up of a callback function (step 5 below) to compute functional values of nonlinear terms, and optionally their gradients (step 6 below), in the model.

For an efficient implementation of the black-box interface, the front-end or calling application must do the following steps:

1. Create a LINDO environment with a call to *LScreateEnv*().
2. Create a model structure in this environment with a call to *LScreateModel*().
3. Load problem structure and linear data into the model structure with a call to *LSloadLPData*().
4. Load nonlinear problem structure into the model structure with a call to *LSloadNLPData*().
5. Provide a pointer to a nonlinear function evaluation routine with a call to *LSsetFuncalc*().
6. Optionally, provide a pointer to a gradient calculation routine with a call to *LSsetGradcalc*().
7. Solve the problem with a call to *LSoptimize*().
8. Retrieve the solution with calls to *LSgetInfo*(), *LSgetPrimalSolution*(), and *LSgetDualSolution*().
9. Delete the model and environment with a call to *LSdeleteEnv*().

We first illustrate with an example how LP and NLP pieces of model data are loaded to the solver. Set up of callback functions referred to in steps 5 and 6 are discussed later in this chapter.

Loading Model Data

Consider the following minimization problem with 8 variables and 6 constraints, with finite bounds on all variables. It has both linear and nonlinear components.

Minimize		x0 + log(x0*x1) + x3 + $x2^2$			
s.t.					
Constraint 0:		+ x3 + x5		≤	400
Constraint 1:		- 25*x3 + 25*x4 + 25*x6		≤	10000
Constraint 2:		- x4 + x7		≤	100
Constraint 3:		100*x0 - x1*x5 + 883*x3		≤	83333
Constraint 4:		x2*x3 - x2*x6 – 1250*x3 + 1250*x4		≤	0.0
Constraint 5:		x3*x4 – x2*x7 – 2500*x4		≤	-1250000
Bounds:					
100	≤	x1		≤	10000
1000	≤	x2		≤	10000
1000	≤	x3		≤	10000
10	≤	x4		≤	1000
10	≤	x5		≤	1000
10	≤	x6		≤	1000
10	≤	X7		≤	1000
10	≤	X8		≤	1000

Phase I: Loading LP structure

The first step in loading model data is to ignore all nonlinear terms in the model and scan for linear terms to construct the LP coefficient matrix.

Examining the model row-by-row, we note that

1. Objective row, indexed -1, is linear in x0 and x3,
2. Constraint 0 is linear in x3 and x5,
3. Constraint 1 is linear in x3, x4, and x6,
4. Constraint 2 is linear in x4 and x7,
5. Constraint 3 is linear in x0 and x3,
6. Constraint 4 is linear in x4,
7. Constraint 5 is not linear in any variables.

Denoting nonlinear coefficients by “*”, this implies the NLP model has the following coefficient matrix

A =

	0	1	2	3	4	5	6	7
0				1		1		
1				-25	25		25	
2					-1			1
3	100	*		883		*		
4			*	*	1250		*	
5			*	*	*			*

The matrix has 8 columns, 6 rows and 19 nonzeros. Using the guidelines in Chapter 1, we obtain the following sparse representation, where we use a 0 as a place-holder for nonlinear coefficients

```
Column-start=[0, 1, 2,    4,                  9,             13,    15,   17,
19]

Row-Index= [  3, 3, 4, 5, 0,   1,   3, 4, 5,  1,  2,    4, 5, 0, 3,  1, 4, 2, 5]

Values   = [100, 0, 0, 0, 1, -25, 883, 0, 0, 25, -1, 1250, 0, 1, 0, 25, 0, 1, 0]
```

Other LP components, the cost vector, right-hand-side values, variable bounds, and constraint senses, are obtained from the original model that can be listed as

```
Objective coefficients = [    1,     0,     0,    1,    0,    0,    0]

Lower bounds           = [  100,   100,  1000,   10,   10,   10,   10]

Upper bounds           = [10000, 10000, 10000, 1000, 1000, 1000, 1000]

Right-hand-side vector = [400, 10000, 100, 83333, 0, -1250000]

Constraint senses      = [  L,     L,   L,     L, L,        L]
```

These Phase-I linear components can be represented using arrays of appropriate types in any programming language and be loaded to LINDO API using the LSloadLPData function just as in a linear model.

Phase II: Loading NLP structure

The next step in loading model data is to ignore all linear terms in the model and determine the nonzero structure of the NLP terms. A nonlinear (nonzero) coefficient is said to exist for row *i,* variable *j*, if the partial derivative of a row *i* with respect to variable *j* is not a constant. Scanning the model row-by-row, we observe the following

8. Objective row, indexed -1, is nonlinear in x0, x1 and x2,
9. Constraint 0 has no nonlinear terms,
10. Constraint 1 has no nonlinear terms
11. Constraint 2 has no nonlinear terms
12. Constraint 3 is nonlinear in x1 and x5,
13. Constraint 4 is nonlinear in x2, x3 and x6,
14. Constraint 5 is nonlinear in x2, x3, x4 and x7

At this point we are interested in only the nonlinear coefficients of the constraints, i.e., the "*" in the previous matrix. The sparse representation of this sub-matrix is

```
Column-start = [0, 0, 1,    3,    5, 6, 7, 8, 9]

Row-Index       = [3, 4, 5, 4, 5, 5, 3, 4, 5]
```

The nonlinearities in the objective function are represented in a similar fashion using sparse representation. We simply determine the number of nonlinear variables in the objective function and place the indices of these nonlinear variables in an array.

```
Number of nonlinear-variables = 3

Nonlinear variable-index = [0, 1, 2]
```

As in phase-I, these components can be represented using arrays of appropriate types in any programming language, and be loaded to the solver via LSloadNLPData function. If required, integrality restrictions can be imposed using LSloadVarType function (see Chapter 2). In the section *Sample Programming Problems*, Examples 1 and 3 give complete code illustrating the Black-box style method.

Evaluating Nonlinear Terms via Callback Functions

The black-box approach requires the user to set up a callback function that computes the functional values for $f(x)$ and $g_i(x)$ for a given a row index i. A reference to this function is passed to the solver via LSsetFuncalc() routine so that it could evaluate functional values as needed. Optionally, a second callback function, which computes the partial derivatives, could be set via LSsetGradcalc() routine. However, since LINDO API is equipped with a versatile differentiation toolbox, it can compute the partial derivatives using functional values provided by the first callback function. This makes the use of a second callback function for derivatives optional. In this approach, if the user does not provide a second callback function, the solver will automatically invoke its internal differentiation tools to compute derivatives.

For certain classes of NLP models, however, a carefully implemented callback function for partial derivatives may be a more efficient than automatic differentiation. In particular, for models where the nonlinear terms have potential numerical issues over certain ranges in the domains they are defined, a user-defined function may provide better means to control numerical accuracy. This advantage could lead to improved overall performance.

In the following, we give the C prototypes for these callback functions. The function names, *pFuncalc*() and *pGradcalc*(), are arbitrary, and are used merely for illustration. Since these functions will reside in your calling application, you may choose any name you wish. However, the interfaces described must be preserved.

pFuncalc()

Description:

This is a user/frontend supplied routine to compute the value of a specified nonlinear row, given a current set of variable values. This function must be provided in order to solve nonlinear programs with black-box style interface. Use the *LSsetFuncalc*() routine (see Chapter 2) to identify your *pFuncalc*() routine to LINDO API.

Returns:

Returns a value greater than 0 if a numerical error occurred while computing the function value (e.g., square root of a negative number). Otherwise, returns 0.

Prototype:

```
int    pFuncalc ( pLSmodel pModel,
                  void    *pUserData,    int    nRow,
                  double  *pdX,          int    nJDiff,
                  double   dXJDiff,      double *pdFuncVal,
                  void    *pReserved);
```

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.
pUserData	Pointer to a user data area or structure in which any data needed to calculate function values can be stored (e.g., the number of variables). LINDO API obtains the value of this pointer when the *pFuncalc*() routine is established through a call to *LSsetFuncalc*() (see below). Subsequently, whenever LINDO API calls your *pFuncalc*() routine, it passes the same pointer value through *pUserData*. Any data that *pFuncalc*() needs to compute the value of a row in the model should be stored in the data structure pointed to by *pUserData*.
nRow	The row of the model to be evaluated. If *nRow* = -1, then it is assumed to be the objective row. Otherwise, *nRow* is the 0-based index of the row to evaluate.
pdX	A pointer to an array containing the values of the decision variables at the point where the row is to be evaluated (i.e., *pdX*[*j*] = value of variable *j* at current point for *j* = 0, 1, …, *nNVars* –1, where *nNVars* is the number of variables).

nJDiff, dXJDiff	If *nJDiff* is < 0, then *pdX*[] contains a new base point. If 0 ≤ *nJDiff* < *nNVars*, then the current point is different from the current base point solely in dimension *nJDiff*, and the value of *pdX*[*nJDiff*] at the base point is contained in the *dXJDiff* variable. If *nJDiff* is ≥ the number of variables, then *pdX*[] contains the previous base point, but the row to evaluate, *nRow*, has changed. Without loss of correctness, *nJDiff* and *dXJDiff* can be ignored (by setting *nJDiff*=-1). In certain cases, however, exploiting these arguments can reduce function evaluation times dramatically (an example is given below). Keep in mind that your implementation will be complicated through the use of these parameters. Therefore, the first time user may choose to ignore them.

Output Arguments:

Name	Description
pdFuncVal	**pdFuncVal* returns the value of the function.
pReserved	A pointer reserved for future use.

Remarks:

- Any explicit constant term is assumed to have been brought to the left-hand side of the constraint. That is, an equality constraint is assumed to have been put in the standard form $g_i(x) = 0$. *pdFuncalc*() returns the value of $g_i(x)$.
- The parameter *nJDiff* allows *pFuncalc*() to exploit some efficiencies in typical usage. In a model with many nonlinear variables, a major portion of the work in *pFuncalc*() may be in copying the variable values from *pdX*[] to local storage (typically in *pUserData*). The nonlinear solver may call *pFuncalc*() several times sequentially where the only difference in inputs is in the parameter *nRow* (i.e., the *pdX*[] values remain unchanged). Values of *nJDiff* ≥ the number of variables indicate this situation.
- Somewhat similarly, if finite differences rather than derivatives are being used, the nonlinear solver may call *pFuncalc*() several times sequentially where the only difference in the *pdX*[] vector is in a single element *pdX*[*nJDiff*]. Thus, if *pFuncalc*() has retained the values of the *pdX*[] from the previous call, then only the value *pdX*[*nJDiff*] need be copied to local storage.
- Further efficiencies may be gained when a row is separable in the variables. For example, suppose the objective is: $\sum_{i=1,1000} (\log(x[i])$. This would be an expensive function to evaluate at each point due to the time required to compute logarithms. In the case where finite differences are being used, performance could be improved dramatically in the case where *pdX*[] differs from the base point in a single dimension (i.e., when 0 ≤ *nJDiff* < number of variables). For example, suppose you have stored the function's value at the base point in the variable *dGBase*, which will typically be part of the *pUserData* structure. This would allow us to recalculate the row's value using the formula: *dGBase* + log(*pdX*[*nJDiff*]) – log(*dXJBase*). This strategy reduces the number of logarithm computations to only 2 rather than 1000.

pGradcalc()

Description:

This is a user-supplied routine to compute the partial derivatives (i.e., gradient) of a specified nonlinear row given a current set of variable values. This function's name, *pGradcalc*(), is arbitrary, and is used merely for illustration. Since this function will reside in your calling application, you may choose any name you wish. However, the interface described below must be duplicated. This function must be provided only if you do not want LINDO API to use finite differences. In which case, *pGradcalc*() will be called by LINDO API when it needs gradient information. Use the *LSsetGradcalc*() routine (see below) to identify your *pGradcalc*() routine to LINDO API.

Returns:

Returns a value greater than 0 if a numerical error occurred while computing partial values (e.g., square root of a negative number). Otherwise, returns 0.

Prototype:

```
int    pGradcalc ( pLSmodel pModel,
                      void    *pUserData,      int    nRow,
                    double    *pdX,         double    pdLB,
                    double    *pdUB,           int    nNewPnt,
                       int     nNPar,          int   *pnParList,
                    double    *pdPartial)
```

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.
pUserData	Pointer to a user data area or structure in which you can store any data that is needed to calculate gradient values (e.g., number of variables). LINDO API obtains the value of this pointer when you establish your *pGradcalc* () routine through a call to *LSsetGradcalc*() (see below). Subsequently, whenever LINDO API calls your *pGradcalc* () routine, it passes the same pointer value through *pUserData*. Any data that *pGradcalc* () needs to compute partial derivatives should be stored in the data structure pointed to by *pUserData*. The *pUserData* data structure may be the same one used in *pFuncalc*().
nRow	The row of the model for which partial derivatives are needed.
pdX	The values of the decision variables at the current point (i.e., *pdX*[*j*] = value of variable *j* at current point, for *j* = 0, 1,…).
pdLB	*pdLB*[*j*] = lower bound on variable *j*.
pdUB	*pdUB*[*j*] = upper bound on variable *j*.
nNewPnt	*nNewPnt* will be 0 if the variable values in *pdX*[ ] are the same as in the preceding call. If these values are still stored in your *pUserData* memory block, then they need not be copied again,

	thereby improving performance. If any of the values are different, then *nNewPnt* will be greater than 0. Without loss of correctness, *nNewPnt* can be ignored. In certain cases, however, exploiting the information it provides can reduce function evaluation time.
nNPar	Number of variables for which partial derivatives are needed.
pnParList	*pnParlList*[*j*] = 0-based index of the *j*-th variable for which a partial derivative is needed.

Output Arguments:

Name	Description
pdPartial	*pdPartial*[*j*] = partial derivative with respect to variable *j*. In most cases, many of the elements of *pdPartial*[] will not have to be set. You need only set those elements listed in *pnParList*[]. LINDO API allocates the space for this array before calling *pGradcalc* ().

Remarks:

- The variable bounds are passed for use in computing partials of functions with discontinuous derivatives. Note, the bounds may change from one call to the next if the model contains integer variables and the solver is performing branch-and-bound.

Instruction-List Style Interface

Under the instruction list style, the front end supplies a set of instruction lists, one instruction list for each row of the model. LINDO API will automatically detect linearity and exploit it. An instruction list is a vector of integers that encodes the original mathematical model. It represents the model in a variant of Reverse Polish notation (also called *postfix* notation). This scheme is attractive in that it is concise, easy to edit, easy and fast to run, and, most important, it retains the original mathematical structure of your model. A model can be loaded in instruction list format with LSloadInstruct( ).

Postfix Notation in Representing Expressions

Expressions in postfix notation consist of two elements: *operators* (e.g., addition and multiplication) and *operands* (e.g., variables or constants). Most operators are binary in the sense that they take two operands. In typical infix mathematical notation, binary operators appear between their operands (e.g., in *A*+*B* the operator '+' comes between its operands *A* and *B*). In postfix or Reverse Polish notation, the operator comes after its operands. Thus, *A*+*B* is expressed *A B* +. There are also some operators that are unary and take a single operand. In this case, the ordinary mathematical notation (e.g., *exp*(*A*)) is transformed into the postfix notation by reversing the sequence (e.g., A *exp*).

In infix notation, there may be ambiguity in the proper order of execution of various operators. This ambiguity is resolved in infix notation by specifying a priority among the operators (i.e., basic mathematical operator precedence). For example, when evaluating 2+6/3, we do the division before the addition and get the answer 4. Thus, the division operator has higher precedence than the addition. As a second example, when evaluating 8-5-2, we evaluate it as (8-5)-2 rather than as 8-(5-2) and get the answer 1. Similarly, 8-5+2 is taken as (8-5)+2, not 8-(5+2). The general rule is that if two adjacent operators are the same, or have equal precedence, then the leftmost takes precedence. There exists a

means to override the precedence rules by employing parentheses. For example, we can write (3+7)/2 if we want the + to be executed before the /. In postfix notation, all ambiguity has been removed and there are no parentheses. The following are examples of postfix notation.

Infix	Postfix
A	*A*
(*A*)	*A*
A/*B*	*A B* /
A+*B***C*	*A B C* * +
(*A*+*B*)**C*	*A B*+ *C* *
A/*B*-*C*/7	*A B* / *C* 7 / -
A-*B*-3	*A B* - 3 -
A+(*B*-*C*/*D*)-*E***F*	*A B C D* / - + *E F* * -

In order to appreciate the usefulness of postfix notation, it helps to understand how it is used. Postfix instructions are executed on a "stack based" pseudo computer. This stack pseudo computer has only two very simple rules of operation:

1. When an operand is encountered, load its value on top of a stack of numbers.
2. When an operator is encountered, apply it to the numbers on top of the stack and replace the numbers used by the result.

Consider the infix expression: 5+6/3. The postfix expression is 5, 6, 3, /, +.

After the first three terms in postfix notation have been encountered, the stack will look like:

.
3
6
5

Postfix Stack

After the "/" is encountered, the stack will look like:

.
2
5

Postfix Stack

After the "+" is encountered, the stack will look like:

.
7

Postfix Stack

This illustrates that after a properly formed postfix expression is executed, the stack will contain only one number. That number is the value of the expression.

For LINDO API, a postfix expression is simply a list of integers. Each operator has a unique integer associated with it. For example, "+" is represented by the integer 1. Each operand is represented by two integers. The first integer effectively says "Here comes an operand". The second integer specifies which operand. For example, x_{23}, is represented by the integer 23. All currently supported operators and their functions are exhibited below, where *A* and/or *B* and/or *C* and/or *D* are argument(s) of each function or operand(s) to the operator. The integer associated with each operator can be found in the *lindo.h* header file that came with LINDO API.

Supported Operators and Functions

A list of currently supported operators and functions are listed in the following table.

Operator	Index	Function	Global supported?	Description of result
EP_NO_OP	0000	--	Y	No operation.
EP_PLUS	1001	*A* + *B*	Y	Addition of *A* and *B*.
EP_MINUS	1002	*A* – *B*	Y	Subtraction of *A* to *B*.
EP_MULTIPLY	1003	*A* * *B*	Y	Multiplication of *A* and *B*.
EP_DIVIDE	1004	A / B	Y	Division of *A* by *B*.
EP_POWER	1005	*A* ^ *B*	Y	Power of *A* to *B*.
EP_EQUAL	1006	*A* = *B*	Y	True(1) if *A* is equal to *B*, else false(0).
EP_NOT_EQUAL	1007	*A* != *B*	Y	True if *A* is not equal to *B*.
EP_LTOREQ	1008	*A* <= *B*	Y	True if *A* is less-than-or-equal-to *B*.
EP_GTOREQ	1009	*A* >= *B*	Y	True if *A* is greater-than-or-equal-to *B*.
EP_LTHAN	1010	*A* < *B*	Y	True if *A* is less than *B*.
EP_GTHAN	1011	*A* > *B*	Y	True if *A* is greater than *B*.
EP_AND	1012	*A and B*	Y	Logic conjunction: the expression is true if *A* and *B* are both true.
EP_OR	1013	*A or B*	Y	Logic disjunction: the expression is true if *A* or *B* are true.
EP_NOT	1014	~ *A*	Y	The logic complement of *A*; 1 if *A* = 0, 0 if *A* > 0
EP_PERCENT	1015	*A* / 100	Y	The percentage of *A*.
EP_NEGATE	1017	- *A*	Y	Negative value of *A*.
EP_ABS	1018	|*A*|	Y	Absolute value of *A*.
EP_SQRT	1019	$(A)^{1/2}$	Y	Square root of *A*.
EP_LOG	1020	*log*(*A*)	Y	Common logarithm (base 10) of *A*.
EP_LN	1021	*ln*(*A*)	Y	Natural logarithm of *A*.
EP_PI	1022	*3.141592653589793*	Y	Load or push onto the top of the stack the ratio of the circumference of a circle to its

				diameter.
EP_SIN	1023	*sin*(*A*)	Y	Sine of *A*(in radians).
EP_COS	1024	*cos*(*A*)	Y	Cosine of *A*(in radians)..
EP_TAN	1025	*tan*(*A*)	Y	Tangent of *A* (in radians).
EP_ATAN2	1026	*atan2*(*A*,*B*)	Y	Inverse arc tangent (in radians) of *A* (i.e., *atan*(*B*/*A*)).
EP_ATAN	1027	*atan*(*A*)	Y	Arc tangent (in radians) of *A*.
EP_ASIN	1028	*asin*(*A*)	Y	Arc sine (in radians) of *A*.
EP_ACOS	1029	*acos*(*A*)	Y	Arc cosine (in radians) of *A*.
EP_EXP	1030	*exp*(*A*)	Y	The constant *e* raised to the power *A*.
EP_MOD	1031	*mod*(*A*,*B*)	Y	Remainder of *A*/*B*.
EP_FALSE	1032	*F*	Y	Load or push a 0 (False) onto the top of the stack.
EP_TRUE	1033	*T*	Y	Load or push a 1 (True) onto the top of the stack.
EP_IF	1034	*if*(*A*,*B*,*C*)	Y	Returns *B*, if *A* is true (!=0) and returns *C*, if *A* is false (=0).
EP_PSN	1035	*psn*(*A*)	Y	Cumulative standard Normal probability distribution, also known as the cumulative distribution function (cdf), i.e., Prob{standard Normal random variable ≤ *A*}.
EP_PSL	1036	*psl*(*A*)	Y	Unit Normal linear loss function (i.e., E[max{0, *Z*-*A*}], where *Z* = standard Normal).
EP_LGM	1037	*lgm*(*A*)	Y	Natural (base *e*) logarithm of the gamma (i.e., *ln*((*A*-1)!) when *A* is a positive integer).
EP_SIGN	1038	*sign*(*A*)	Y	-1 if A<0, +1 if A >0, else 0
EP_FLOOR	1039	$\lfloor A \rfloor$	Y	Integer part of *A* when fraction is dropped. E.g., floor(-4.6) = -4.
EP_FPA	1040	*fpa*(*A*,*B*)	Y	Present value of an annuity (i.e., a stream of $1 payments per period at interest rate of *A* for *B* periods starting one period from now).

EP_FPL	1041	*fpl*(*A*,*B*)	Y	Present value of a lump sum of $1 *B* periods from now if the interest rate is *A* per period. Note, *A* is a fraction, not a percentage.
EP_PEL	1042	*pel*(*A*,*B*)	Y	Erlang's loss probability for a service system with *B* servers and an arriving load of *A*, no queue allowed.
EP_PEB	1043	*peb*(*A*,*B*)	Y	Erlang's busy probability for a service system with *B* servers and an arriving load of *A*, with infinite queue allowed.
EP_PPS	1044	*pps*(*A*,*B*)	Y	Cumulative Poisson probability distribution. It returns the probability that a Poisson random variable, with mean value *A*, is less-than-or-equal-to *B*.
EP_PPL	1045	*ppl*(*A*,*B*)	N	Linear loss function for the Poisson probability distribution. It returns the expected value of *max*(0,*Z*-*B*), where *Z* is a Poisson random variable with mean value *A*.
EP_PTD	1046	*ptd*(*A*,*B*)	N	Cumulative distribution function for the *t* distribution with *A* degrees of freedom. It returns the probability that an observation from this distribution is less-than-or-equal-to *B*.
EP_PCX	1047	*pcx*(*A*,*B*)	N	Cumulative distribution function for the *Chi*-squared distribution with *A* degree of freedom. It returns the probability that an observation from this distribution is less-than-or-equal-to *B*.
EP_WRAP	1048	*wrap*(*A*,*B*)	Y	Transform A into the interval [1, *B*], If *A* > *B*, then *A* is "wrapped around". E.g., WRAP(14,12)= 2. More generally, =1+mod(*A*,*B*-1).
EP_PBN	1049	*pbn*(*A*,*B*,*C*)	N	Cumulative binomial probability. It returns the probability that a sample of *B* items, from a universe with a fraction of *A* of those items defective, has *C* or

				less defective items.
EP_PFS	1050	*pfs*(*A*,*B*,*C*)	N	Expected number of customers waiting for repair in a finite source Poisson service system with *B* servers in parallel, *C* customers, and a limiting load of *A*.
EP_PFD	1051	*pfd*(*A*,*B*,*C*)	N	Cumulative distribution function for the *F* distribution with *A* degrees of freedom in the numerator and *B* degrees of freedom in the denominator. It returns the probability that an observation from this distribution is less-than-or-equal-to *C*.
EP_PHG	1052	*phg*(*A*,*B*,*C*,*D*)	N	Cumulative hyper geometric probability. it returns the probability that *D* or fewer items in the sample are good, given a sample without replacement of *C* items from a population size *A* where *B* items in the population are good.
EP_RAND	1053	*rand*(*A*)	N	Pseudo-random number uniformly distributed between 0 and 1, depending deterministically on the seed *A*.
EP_USER	1054	*user*(*user_specified _arguments*)	N	Returns the value of the function supplied by the user through *LSsetUsercalc*. The operator is followed by an integer specifying the number of arguments, and preceded by the arguments. See the definition of Usercalc() function below for a prototype.
EP_SUM	1055	*sum*(A_1 , …, A_n)	Y	Summation of vector *A* (i.e., A_1 + … + A_n). The operator is followed by an integer specifying the vector length *n*, and then the vector itself.
EP_AVG	1056	*avg*(A_1 , …, A_n)	Y	Average of vector *A* (i.e., (A_1 + … + A_n) / *n*). The operator is followed by an integer specifying the vector length *n*, and then the vector itself.
EP_MIN	1057	*min*(A_1 , …, A_n)	Y	Minimum value in vector *A*. The

				operator is followed by an integer specifying the vector length *n*, and then the vector itself.
EP_MAX	1058	$max(A_1, \ldots, A_n)$	Y	The maximum value in vector *A*. The operator is followed by an integer specifying the vector length *n*, and then the vector itself.
EP_NPV	1059	$npv(A,B_1,\ldots,B_n)$	N	Net present value of an investment, using a discount rate (*A*) and a series of future payments ($B_1,\ldots,B_n$). The operator is followed an integer specifying the vector length, which is the number of payments n plus 1.
EP_VAND	1060	A_1 *and* A_2*...and* A_n	Y	Vector AND: Returns true if the vector *A* are all true. The operator is followed by an integer specifying the vector length *n*, and then the vector itself.
EP_VOR	1061	A_1 *or* A_2*...or* A_n	Y	Vector OR: Returns true if there is at least one element in the vector A that is true. The operator is followed by an integer specifying the vector length *n*, and then the vector itself.
EP_PUSH_NUM	1062	*A*	Y	Load or push a constant *A* onto the top of stack.
EP_PUSH_VAR	1063	*A*	Y	Load or push a variable *A* onto the top of stack.
EP_NORMDENS	1064	*Normdens(A)*	Y	Standard Normal density at A, also known as the probability density function (pdf), i.e., $(\exp(-A*A/2))/((2*\pi)^{0.5})$.
EP_NORMINV	1065	*NormInv(A, B, C)*	N	Inverse of the cumulative Normal distribution with input probability *A,* mean *B* and standard deviation *C*.
EP_TRIAINV	1066	*TriaInv(A, B, C, D)*	N	Inverse of a triangular cumulative distribution with input probability *A,* for lowest possible value *B*, mode *C*, and highest value *D*.
EP_EXPOINV	1067	*ExpoInv(A, B)*	N	Inverse of an exponential with input probability *A* and mean *B*, i.e. $-B*\ln(1-A)$.

EP_UNIFINV	1068	*UnifInv(A, B, C)*	N	Inverse of a uniform cumulative distribution *A* for supplied low *B* and high *C* values.
EP_MULTINV	1069	*MultInv(A, B_1,..., B_n, C_1,..., C_n)*	N	Inverse of a multinomial cumulative distribution. *A* for a supplied probability vector *B* and corresponding value vector *C*. The operator is followed by an integer specifying the vector length, which is 2n+1, where n is the vector length of B and C.
EP_USRCOD	1070	*UserCode ndx*	Y	A user-defined instruction code. It is treated as an EP_NO_OP along with the integer immediately following it in the list.
EP_SUMPROD	1071	*SumProd*(V_1 , V_2 , V_3 ,.. , V_n)	Y	Vector inner product. Multiplies corresponding components in each vector, and returns the sum of those products (i.e., v_{11}* v_{21}*…*v_{m1} + v_{12}* v_{22}*…*v_{m2} + … + v_{1n}* v_{2n}*…*v_{mn}). Note that these *n* vectors must have the same length of *m*. The operator is followed by two integers, specifying the number of vectors *n* and the vector length *m*, respectively. The syntax is: V_1 , V_2 , V_3 ,.. , V_n, *EP_SUMPRROD, n, m.*
EP_SUMIF	1072	*SumIf(w, V_1 , V_2)*	Y	This vector type of operator adds the component in vector V_2, if its corresponding component in vector V_1 is equal to the target *w* (i.e., *if(w==* v_{11} , v_{21} *,0) + if(w==* v_{12} , v_{22} ,0) + … + *if(w==* v_{1n} , v_{2n} ,0)). Note, both vectors must have the same length of *n*. The operator is followed by an integer, specifying the number of vector *n*. The syntax is: *w*, V_1 , V_2 , *EP_SUMIF, n.*
EP_VLOOKUP	1073	*Vlookup*(*w*, V_1 , V_2, *range_logic*)	Y	This vector type of operator searches for a component in the first vector V_1 with respect to the target *w*, and then returns the

				corresponding component in the second vector V_2. The *range_logic*, which takes a value of 0 and 1 for False or True case, respectively, decides which type of logic used to select the winner. When *range_logic* is False, it returns: *if*($w==v_{11}$, v_{21}, *if*($w==v_{12}$, v_{22},..., *if*($w==v_{1n}$, v_{2n},, *Infinity*))). When *range_logic* is False, it returns: *if*($w<v_{11}$, *Infinity*, *if*($w<v_{12}$, v_{21},..., *if*($w<v_{1n}$, $v_{2(n-1)}$, v_{2n}))). Note that both vectors must have the same length of *n*. The operator is followed by two integers, specifying the vector length *n* and *range_logic*, respectively. The syntax is: *w*, V_1, V_2, *EP_VLOOKUP*, *n*, *range_logic*.
EP_VPUSH_NUM	1074	$n_1, n_2, n_3,...,n_m$	Y	Vector Push Number. Loads a vector of number indices $n_1, n_2, n_3,...,n_m$. The operator is followed by an integer, specifying the vector length *m*. The syntax is: n_1, n_2, n_3,... , n_m, *EP_VPUSH_NUM*, *m*.
EP_VPUSH_VAR	1075	$v_1, v_2, v_3,...,v_m$	Y	Vector Push Variable. Loads a vector of variable indices $v_1, v_2, v_3,...,v_m$. The operator is followed by an integer, specifying the vector length *m*. The syntax is: $v_1, v_2, v_3,...,v_m$, *EP_VPUSH_VAR*, *m*.
EP_VMULT	1074	$A_1 * A_2 * \ldots * A_m$	Y	This vector type of operator sequentially multiplies each element in vector A. The operator is followed by an integer, specifying the vector length *m*. The syntax is: v_1, v_2, v_3,... , v_m, *EP_VMULT*, *m*.
EP_SQR	1077	A^2	Y	Square of A.
EP_SINH	1078	*Sinh*(*A*)	Y	Hyperbolic sine of *A*.
EP_COSH	1079	*Cosh*(*A*)	Y	Hyperbolic cosine of *A*.
EP_TANH	1080	*Tanh*(*A*)	Y	Hyperbolic tangent of *A*.
EP_ASINH	1081	$Sinh^{-1}(A)$	Y	Inverse hyperbolic sine of *A*.

EP_ACOSH	1082	$Cosh^{-1}(A)$	Y	Inverse hyperbolic cosine of A .
EP_ATANH	1083	$Tanh^{-1}(A)$	Y	Inverse hyperbolic tangent of A .
EP_LOGB	1084	$Log_B(A)$	Y	Logarithm of A with base B.
EP_LOGX	1085	$A * Log(A)$	Y	A times common logarithm (base 10) of A .
EP_LNX	1086	$A * Ln(A)$	Y	A times natural logarithm of A.
EP_TRUNC	1087	*Trunc*(A, B)	Y	Truncates A to a specified precision of B by removing the remaining part of value A .
EP_NORMSINV	1088	*NormSInv*(A)	Y	Inverse of the cumulative standard Normal distribution with input probability A.
EP_INT	1089	*Int*(A)	Y	Largest integer $\leq A$. E.g., int(-4.6) = -5, and int(4.6) = 4.
EP_PUSH_STR	1090	*string*(A)	Y	Push string in position A of strings loaded with LS_load_string.
EP_VPUSH_STR	1091	$string_1$, $string_2$, …,$string_m$.	Y	Push a vector of strings. The operator is followed by an integer, specifying the vector length m. The syntax is: $string_1$, $string_2$,…, $string_m$, *EP_VPUSH_STR*, m.
EP_PUSH_SPAR	1092	A	Y	Load or push a stochastic (random) parameter A onto the top of stack.
EP_NORMPDF	1093	A	Y	Probability density function of the standard Normal distribution evaluated at A.
EP_NORMCDF	1094	A	Y	Cummulative density function of the standard Normal distribution evaluated at A.

Grey-Box Style Interface

The grey-box style interface allows the user to supply some functions in instruction list style and others in the black-box style. This mixed approach is particularly useful for cases where function evaluating routines were implemented in the past (possibly in some other language) and it is imperative to reuse the existing source code. It might also be that case that some functions are difficult to express in an instruction list or even impossible due to lack of closed forms (like simulation output). In such case, the user can formulate an instruction-list using the EP_USER operator wherever the need arises to evaluate some of the expressions in a user-defined function. A simple C programming example using the grey-box interface is given as Example 5.

Usercalc()

Description:

In Grey-box style interface, this is the user/front-end supplied routine, required by the EP_USER operator, to compute the value of a user-defined function for a given set of arguments. The arguments the function uses are passed through in a double array of a pre-specified size.

This function name used here, *Usercalc*(), is arbitrary, and is merely for illustration. Since this function will reside in your calling application, you may choose any name you wish. However, the interface described below must be duplicated.

This function should be provided for all nonlinear models that contain the EP_USER operator. This operator is particularly useful in expressing nonlinear relations, which are difficult or impossible to express in closed form. You should use the *LSsetUsercalc*() routine to identify your *Usercalc*() routine to LINDO API.

Returns:

Returns a value greater than 0 if a numerical error occurred while computing the function value (e.g., square root of a negative number). Otherwise, return 0.

Prototype:

int	Usercalc (pLSmodel pModel, int nArgs, double *pdValues, void *pUserData, double *pdFuncVal);

Input Arguments:

Name	Description
pModel	Pointer to an instance of *LSmodel*.
nArgs	The number of arguments the function requires to evaluate the function value.
pdValues	A pointer to a double array containing the values of the arguments that will be used to evaluate the function. The size of this array is specified by *nArgs*.
pUserData	Pointer to a user data area or structure in which any other data needed to calculate function values can be stored (e.g., input for a simulation experiment). LINDO API obtains the value of this pointer when the *Usercalc*() routine is established through a call to *LSsetFuncalc*() (see below). Subsequently, whenever LINDO API calls your *Usercalc*() routine, it passes the same pointer value through *pUserData*. Any data that *Usercalc*() needs to compute the function value could be stored in the data structure pointed to by *pUserData.*

Output Arguments:

Name	Description
pdFuncVal	**pdFuncVal* returns the value of the function.

Remark:

This single Usercalc() function can be used as a gateway to multiple black-box functions by extending the use of pdValues array. You simply increase the number of arguments by 1 and use the first argument (pdValues[0]) as an integer to identify which black-box function needs to be computed. In such a framework, each EP_USER instruction-block will have to be extended with EP_PUSH_NUM operator to include a unique function identifier to pass to Usercalc(). This allows the user to transform Usercalc() into a wrapper for all black-boxes. With the use of "if-else" blocks based on the value of pdValue[0] in Usercalc(), the user can divert the computations to the desired black-box. This approach is illustrated in Example 5 with two black-boxes.

Instruction Format

The instruction for most functions consists of a single operator that is represented by its operator name or the integer referring to its index number in the above list. For example, addition can be written as 'EP_PLUS' or '1' in the instruction list. The exceptions are functions involving a vector argument, such as *EP_SUM*, *EP_AVG*, *EP_MIN*, and *EP_MAX*, or two arguments, such as *EP_SUMPROD* and *EP_VLOOKUP*. Here an additional integer appears immediately after the operator code in order to specify the number of elements in the operand vector. When the argument is a variable, *PUSH_VAR* is used to invoke loading of the variable and then the position of that variable in the integer vector is specified. For example, variable 'x_0' that is the first variable (at position 0, since LINDO API uses zero-based counting) in the variable vector 'x', takes a vector of [*EP_PUSH_VAR*, 0] or [63, 0] in the instruction list. When the argument is a (double precision) constant, the operator *EP_PUSH_NUM* is used to invoke the loading of the double precision number and then the position of that double precision number in the double precision number vector is specified. For example, say 3.0 is the second number (at position 1) in the double precision number vector of r[5]=[1.0, 3.0, 5.0 2.0, 7.0]. Write [*EP_PUSH_NUM*, 1] or [62, 1] in the instruction list to denote the double precision number 3.0.

Given these representation rules and postfix notation, an instruction list for arbitrary mathematical expressions can now be constructed. Below are three examples to illustrate this translation.

Example 1

Infix expression = $x_0 + x_1 * x_2$. The corresponding postfix expression = [$x_0\, x_1\, x_2\, *\, +$].

If the variable vector is defined as $\mathbf{x} = [x_0, x_1, x_2]$, then the resulting instruction list looks like:

[*EP_PUSH_VAR*, 0, *EP_PUSH_VAR*, 1, *EP_PUSH_VAR*, 2, *EP_MULTIPLY*, *EP_PLUS*]

or, in the equivalent all integer form:

[1063, 0, 1063, 1, 1063, 2, 1003, 1001]

Example 2

Infix expression = $2 * max(x_0, x_1 + 3, sin(x_0 + x_1))$

Notice *max* takes a vector argument with 3 elements. Define a vector of variables $\mathbf{x} = [x_0 , x_1]$ and declare a vector of double precision constants r=[2.0 , 4.0] storing number values. Then, the mathematical expression can be translated into the postfix notation, and the result looks like:

[2 x_0 x_1 3 + x_0 x_1 + *sin* *max* *]

This can be converted to the following instruction list:

[EP_PUSH_NUM, 0, EP_PUSH_VAR, 0, EP_PUSH_VAR, 1, EP_PUSH_NUM, 1, *EP_PLUS*, *EP_PUSH_VAR*, 0, *EP_PUSH_VAR*, 1, *EP_PLUS*, *EP_SIN*, *EP_MAX*, 3, *EP_MULTIPLY*]

or, in the equivalent all integer form:

[1062, 0, 1063, 0, 1063, 1, 1062, 1, 1001, 1063, 0, 1063, 1, 1001, 1023, 1058, 3, 1003]

Example 3

Infix expression= $(x_0 + x_1 - 1) * (x_1 + 3) - 9 * exp(-5 * x_0^2 / x_1)$

Define the vector of variables $\mathbf{x} = [x_0 , x_1]$ and declare a double precision vector r =[1.0 , 3.0 , 9.0, 5.0, 2.0] that includes all double precision numbers in the expression. The corresponding postfix =

[x_0 x_1 + 1 - x_1 3 + * 9 5 x_0 2 ^ * x_1 / - *exp* * -]

Thus, the resulting instruction list looks like:

[EP_PUSH_VAR, 0, EP_PUSH_VAR, 1, EP_PLUS, EP_PUSH_NUM, 0, *EP_MINUS*, *EP_PUSH_VAR*, 1, *EP_PUSH_NUM*, 1, *EP_PLUS*, *EP_MULTIPLY*, *EP_PUSH_NUM*, 2, *EP_PUSH_NUM*, 3, *EP_PUSH_VAR*, 0, *EP_PUSH_NUM*, 4, *EP_POWER*, *EP_MULTIPLY*, *EP_PUSH_VAR*, 1, *EP_DIVIDE*, *EP_NEGATE*, *EP_EXP*, *EP_MULTIPLY*, *EP_MINUS*],

or, in the equivalent all integer form:

[1063, 0, 1063, 1, 1001, 1062, 0, 1002, 1063, 1, 1062, 1, 1001, 1003, 1062, 2, 1062, 3, 1063, 0, 1062, 4, 1005, 1003, 1063, 1, 1004, 1017, 1030, 1003, 1002].

Note that the last operator, " - ", is a negate operation, rather than a minus, because it only involves a single operand in the calculation. Also note that the power expression, [x_0 2 ^], can be equivalently replaced by [x_0 *square*].

Information about the instruction lists and variable bounds are then passed to LINDO API with a call to *LSloadInstruct*.

Differentiation

When solving a general nonlinear problem, the solution method used by LINDO API requires the computation of derivatives. The accuracy and efficiency of the derivative computation are of crucial importance for convergence speed, robustness, and precision of the answer. The instruction list form of input supports two approaches to compute derivatives: finite differences and automatic differentiation. The finite differences approach is used by default as the basic scheme to generate derivatives. For highly nonlinear cases, this approach may have poor numerical precision for computing the matrix of partial derivatives of the constraints, the so-called Jacobian. The automatic differentiation approach computes derivatives directly from the instruction list code. To select the automatic differentiation option, call *LSsetModelIntParameter*() to set the value of parameter *LS_IPARAM_NLP_AUTODERIV* to 1.

Solving Non-convex and Non-smooth models

The two main reasons that you may not have gotten the best possible solution for your nonlinear model are a) the model contained non-convex relations, or b) the model contained nonsmooth relations. Qualitatively, if a model is non-convex, it means that a solution method that moves only in a direction of continuous improvement will not necessarily lead one to the best possible solution. An example of a non-convex model is:

Maximize $(x - 5)^2$;
$0 \le x \le 12$;

If you start at $x = 4$, moving in the direction of an improving solution will lead to a local optimum of $x = 0$. The global optimum is in fact at $x = 12$.

In a nonsmooth model, even though the model is convex, it may be difficult to find a direction of improvement. Examples of nonsmooth functions are abs(x), and max(x, y). For example, abs(x) is not smooth at $x = 0$, while max(x, y) has a sharp break at $x = y$.

An example of a convex and nonsmooth model is:

Minimize max(abs(x-5), abs(y-5));

The obvious global optimum occurs at $x = y = 5$. If you start at $x = y = 0$, the objective value is 5. Notice that increasing x by itself does not help. Decreasing x hurts. Similar comments apply to y. Thus, traditional solution methods based on derivatives may be unable to find a direction of improvement at a point such as $x = y = 0$, which is not even a local optimum. In this case, the solver will simply quit.

LINDO API has three methods available for eliminating difficulties caused by nonsmooth or non-convex functions: a) linearization, b) multiple start points, and c) rigorous global optimization.

Linearization

Using the first of these methods, a process referred to as *linearization*, some of the nonlinear functions and expressions supported by LINDO API may be automatically converted into a series of linear expressions by the solver. Linearization replaces a nonlinear function or expression with a collection of additional variables and linear constraints such that the modified model is mathematically equivalent to the original. However, the nonlinear functions or expressions have been eliminated. Note that the linearization process may internally add a considerable number of constraints and variables, some of which are binary, to the mathematical program generated to optimize your model.

Nonlinear functions, operators, and expressions that may be eliminated through linearization are:

Functions	Operators	Expressions
EP_ABS	<	$x^* y$ (where at least one of x and y is a binary 0/1 variable)
EP_AND	≤	$u^* v = 0$
EP_IF	<>	$u^* v \le 0$
EP_MAX	=	$u^* v \le 0$
EP_MIN	<	
EP_NOT	≥	
EP_OR		

To select the linearization options, you can use *LSsetModelIntParameter*() to set the value of the *LS_IPARAM_NLP_LINEARZ* parameter and determine the extent to which LINDO API will attempt to linearize models. The available options here are:

1. 0 (Solver decides) - Do Maximum linearization if the number of variables is 12 or less. Otherwise, no linearization will be performed.
2. 1 (None) - No linearization occurs.
3. 2 (Minimum)- Linearize *EP_ABS*, *EP_MAX*, and *EP_MIN* functions and expressions of $x^* y$, $u^* v = 0$, $u^* v \le 0$, and $u^* v \ge 0$.
4. 3 (Maximum) - Same as Minimum plus linearize *EP_IF*, *EP_AND*, *EP_OR*, *EP_NOT*, and all logical operations (i.e., ≤, =, ≥, and <>).

By default, this parameter is set to 0 (*Solver decides*).

When a nonlinear model can be fully linearized using nonlinear-to-linear conversions, you may find a global optimum rather than a local minimum, find a solution where none could be found before, and/or find an optimal solution faster. Even when the nonlinear model is merely partially linearized and remains nonlinear after linearization, you still may have a good chance to get the aforementioned benefits. However, there is no mathematical guarantee.

To check the linearity of the model, you can use *LSgetModelIntParameter*() to get the value of the *LS_IPARAM_NLP_LINEARITY* parameter. If the return value is 1, then the solver has determined that your model is linear or has been completely linearized in the pre-processing step. This also means that the solution obtained is the global optimum. If the return value is 0, your model is nonlinear or remains nonlinear after linearization and the solution may be a local optimum.

Note: It is not possible to linearize a model, which is already loaded. If linearization needs to be used, it should be turned on before the call to LSloadInstruct.

Delta and Big M Coefficients

In linearization, two coefficients, *Delta* and *Big M*, are used to build up the additional linear constraints added as part of linearization. The *Delta coefficient* is a measure of how closely the additional constraints should be satisfied. To define the *Delta coefficient*, you can use *LSsetModelDouParameter*() to set the value of the *LS_DPARAM_MIP_DELTA* parameter. LINDO API defaults to the tightest possible *Delta coefficient* of Big M.

On the other hand, when LINDO API linearizes a model, it will add *forcing constraints* to the mathematical program to optimize your model. These forcing constraints are of the form:

$$f(variables) \le M * y$$

where *M* is the *Big M coefficient* and *y* is a 0/1 binary variable. The idea is that, if some activity in the model is occurring, the forcing constraint will drive *y* to take on the value of 1. Given this, setting the *Big M* value too small could result in an infeasible model. The astute reader might conclude it would be smart to make *Big M* quite large, thereby minimizing the change of an infeasible model. Unfortunately, setting *Big M* to a large number can lead to numerical round-off problems in the solver that result in infeasible or suboptimal solutions. Therefore, getting a good value for the *Big M coefficient* may take some experimenting. To set the *Big M coefficient*, use *LSsetModelDouParameter*() to set the value of the *LS_DPARAM_MIP_ LBIGM* parameter. The default value for *Big M* is 1.0e+5.

Precedence Relations and Complementarity Constraints

When the linearization option is turned on, LINDO API will recognize the expression pattern of $x*y$ that involves the multiplication of at least one 0/1 variable (i.e., y). The linearization manager may attempt to rearrange the sequence of a series of products and determine the best linearization strategies. Even when the original model expression involves parentheses to override the precedence rules, the linearization manager still recognizes the possible $x*y$ pattern exists by expanding parentheses. Subsequently, it will linearize these linearizable expressions.

Currently, the linearization manager rearranges and detects products involving only single variables and constants (e.g., $2* x_0* y_0* x_1* y_1$) and performs a comprehensive linearization. Products involving nonlinear elements (e.g., *exp*(x) or *sin(x), x*^2) in complicated expressions (e.g., $2* x_0* y_0*\exp(x_1)$) won't be rearranged. Thus, the model might be merely partially linearized.

For complementarity constraints (i.e., $u*v = 0$ or $u*v \le 0$ or $u*v \ge 0$), LINDO API can recognize and linearize any product of two continuous or discrete variables (i.e., u and v) constrained to be equal to, less than, or greater than zero. In order to be recognized as complementarity constraints, the corresponding instruction list of the constraint should be expressed exactly as:

[*EP_PUSH_VAR*, *(variable index* 1*)* , *EP_PUSH_VAR*, (*variable index* 2), *EP_MULTIPLY*].

Solving and Retrieving the Solution of a Linearized Model

Linearization adds binary variables to the original model and makes it an (mixed) integer (nonlinear) programming problem. In order to ensure proper solution, LSsolveMIP should be run on the linearized model. Solution vectors in a linearized model should be accessed via MIP specific solution query functions (e.g. LSgetMIPPrimalSolution).

Since linearization modifies the original model by adding new variables and constraints to it, the user should be careful when allocating space for solution vectors. In particular, the number of variables and constraints in a linearized model would constitute the basis for the size of solution vectors. For example, a model that has *n_vars* variables without linearization would have *n_vars* + *k* variables with linearization. Similarly, a model that has *n_cons* constraints without linearization would have *n_cons* + *t* constraints with linearization.

The actual values for the number of variables and constraints should be obtained by calling LSgetInfo function and then be used to allocate sufficient space for the solution vectors. The values of the first *n_vars* (*n_cons*) elements in the primal (dual) solution vectors of the linearized model refer to the primal (dual) solution of the original model.

Multistart Scatter Search for Difficult Nonlinear Models

In many real-world systems, the governing dynamics are highly nonlinear and the only way they can be accurately modeled is by using complex nonlinear relationships. Although linear or convex approximations are often possible, there are cases where such approximations lead to a significant loss in the accuracy of the model. In the presence of such nonlinear relationships, the analyst faces the difficult task of solving non-convex nonlinear models. The difficulty is mainly due to three factors: (i) there are an unknown number of locally optimal solutions in the feasible solution set, (ii) the quality of these local solutions vary substantially, and (iii) exploring the solution space, even for small problems, could be prohibitive.

In solving non-convex models, the ultimate goal is to find the best of the local optimal solutions. This is referred to as the global optimum. The optimization task involved with finding the global optimum is called global optimization. In the context of minimization, LINDO API provides globally optimal solutions to linear or convex quadratic (mixed-integer) models. For nonlinear models, the solution returned will be a local optimum and is not known to be the global minimizer. If the nonlinear objective function and the feasible solution set is known to be convex, then any local optimal solution could be assessed as the global minimizer. However, it is generally not possible to check if the nonlinear model under consideration is convex or not. Verifying this is harder than finding a proven global minimizer.

For non-convex nonlinear models, LINDO API is equipped with a global optimization heuristic called the multistart nonlinear solver. This method explores the feasible solution space in search of better local optimal solutions. A multistart method for global optimization refers to a generic algorithm that attempts to find a global solution by starting the main nonlinear solver from multiple starting points in the solution space. This method is stochastic in nature and ensures that the chances to achieve a global optimum are 100% if the algorithm is run indefinitely long. However, for practical purposes, LINDO API allows the user to set an upper limit on the number of local solutions to be examined within a fixed number of iterations or during a finite duration of time. This approach generally leads to locating several high quality local optima and then returns the best one found.

In Figure 7.1, a box-constrained non-convex nonlinear model is illustrated. This is based on a non-convex combination of three Gaussian distributions. The formal model statement is as follows:

```
MINIMIZE Z =  3*(1-X)^2*exp(-(X^2)-(Y+1)^2) - 10*(X/5-X^3-Y^5)*exp(-(X^2)
              -Y^2) - exp(-((X+1)^2)-Y^2)/3
S.T.          3 ≥  X ≥ -3 ,  3 ≥  Y  ≥ -3
```

This model has multiple local optimal solutions and its objective values are highly scale-dependent. In the following section, the section *Example 1: Black-Box Style Interface* below demonstrates how the standard nonlinear solver is used to solve the model. In the *Example 3: Multistart Solver for Non-Convex Models* below, the same model is solved using the multistart solver to demonstrate the achievable improvements in the solution quality. *Example 3* illustrates the use of a standard callback function to access every local solution found during optimization.

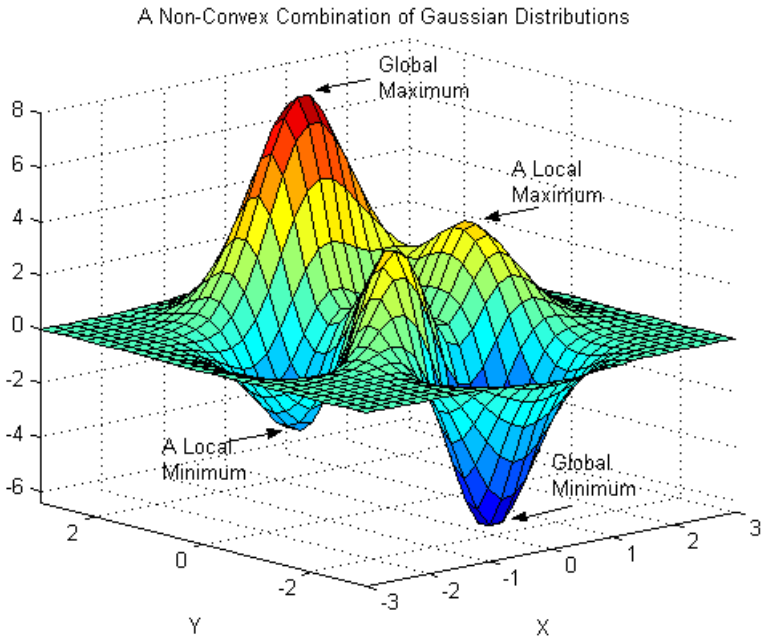


Figure 7.1

Global Optimization of Difficult Nonlinear Models

For difficult nonlinear models that are either non-smooth or non-convex, the multistart search option is worth considering. However, the multistart option does not provide a guarantee of global optimality. If a guarantee of global optimality is desired, then one may invoke the global optimizer in LINDO API. The global optimizer uses a) branching to split the feasible region into sub regions and b) bounding to get a valid bound on the optimal objective value in each sub region. Sub regions for which the bound is worse than some incumbent solution are discarded. A promising sub region may be subdivided further in order to get a more accurate bound. The multistart option works with either the black-box or instruction list style of input. The global optimizer option works only with the instruction list input format. See the sections *Black-Box Style Interface* and *Instruction-List Style Interface* above for more information.

The global solver supports a wide range of mathematical functions. Functions currently supported are identified in the earlier table: “Supported Operators and Functions” in the column, “Global supported”.

If the model contains functions that are not supported, the global solver will terminate without computing a solution and return an error message of LSERR_GOP_FUNC_NOT_SUPPORTED. In such cases, the standard or multistart NLP solvers could be invoked by calling LSoptimize() (or LSsolveMIP() for integer models) to obtain a local optimal solution

An obvious question is, why not use the global solver option all the time? The answer is that finding a guaranteed global optimum is an NP-hard task. That is, just as with integer programs, the time to find a guaranteed global optimum may increase exponentially with problem size.

Sample Nonlinear Programming Problems

Example 1: Black-Box Style Interface:

This example illustrates the use of LINDO API to build and solve a small nonlinear model whose unconstrained version is illustrated in Figure 7.1 above. The black-box style interface is used. This requires a (callback) function to evaluate the objective function and constraints of the model. The callback function will be installed using the *LSsetFuncalc*() routine. A second callback function that computes the partial derivatives of the objective function and constraints is also provided. This second callback function is optional and need not be specified. LINDO API can approximate the derivatives from the functional values using a technique called finite differences.

```
/*
############################################################
#                       LINDO-API
#                    Sample Programs
#                  Copyright (c) 2007 by LINDO Systems, Inc
#
#         LINDO Systems, Inc.           312.988.7422
#         1415 North Dayton St.         info@lindo.com
#         Chicago, IL 60622             http://www.lindo.com
############################################################
  File   : ex_nlp1.c
  Purpose: Solve a NLP using the black-box style interface.
  Model  : A nonlinear model with multiple local minimizers.

        minimize  f(x,y) =  3*(1-x)^2*exp(-(x^2) - (y+1)^2)
                          - 10*(x/5 - x^3 - y^5)*exp(-(x^2)-(y^2))
                          - 1/3*exp(-(x(+1)^2) - (y^2));
        subject to
                          x^2 + y   <=  6;
                          x   + y^2 <=  6;
*/
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "lindo.h"

/* the summands of the objective function */
#define  g1(X,Y) ( exp( -pow(X  ,2)  - pow(Y+1,2) )  )
#define  g2(X,Y) ( exp( -pow(X  ,2)  - pow(Y  ,2) )  )
#define  g3(X,Y) ( exp( -pow(X+1,2)  - pow(Y  ,2) )  )
#define  f1(X,Y) ( pow(1-X,2)                        )
#define  f2(X,Y) ( X/5 - pow(X  ,3)  - pow(Y  ,5)    )
/* partial derivatives of the summands */
#define dxg1(X,Y)  ( g1(X,Y)*(-2)*X     )
#define dyg1(X,Y)  ( g1(X,Y)*(-2)*(Y+1) )
#define dxg2(X,Y)  ( g2(X,Y)*(-2)*X     )
#define dyg2(X,Y)  ( g2(X,Y)*(-2)*Y     )
#define dxg3(X,Y)  ( g3(X,Y)*(-2)*(X+1) )
#define dyg3(X,Y)  ( g3(X,Y)*(-2)*Y     )
#define dxf1(X,Y)  ( 2*(1-X)            )
#define dyf1(X,Y)  ( 0                  )
#define dxf2(X,Y)  ( 1/5 - 3*pow(X,2)   )
#define dyf2(X,Y)  ( -5*pow(Y,4)        )
```

```
/*************************************************************
   Standard callback function to display local and intermediate
   solutions
 *************************************************************/
int  CALLTYPE print_log(pLSmodel model, int iLoc, void *cbData)
{
  int iter=0,niter,biter,siter;
  int *nKKT = (int *) cbData, npass;
  double pfeas=0.0,pobj=0.0,dfeas=0.0;
  double bestobj;
  static int ncalls = 0;

  if (iLoc==LSLOC_LOCAL_OPT)
  {
    LSgetCallbackInfo(model,iLoc,LS_IINFO_NLP_ITER,&niter);
    LSgetCallbackInfo(model,iLoc,LS_IINFO_SIM_ITER,&siter);
    LSgetCallbackInfo(model,iLoc,LS_IINFO_BAR_ITER,&biter);
    LSgetCallbackInfo(model,iLoc,LS_DINFO_POBJ,&pobj);
    LSgetCallbackInfo(model,iLoc,LS_DINFO_PINFEAS,&pfeas);
    LSgetCallbackInfo(model,iLoc,LS_DINFO_DINFEAS,&dfeas);
    LSgetCallbackInfo(model,iLoc,LS_IINFO_MSW_PASS,&npass);
    LSgetCallbackInfo(model,iLoc,LS_DINFO_MSW_POBJ,&bestobj);
    iter = niter+siter+biter;
    printf("%5d %5d %16.5e %16.5e %16.5e %16.5e\n",
      npass,iter,pobj,pfeas,dfeas,bestobj);
    (*nKKT)++;
  }
  else if (iLoc == LSLOC_CONOPT)
  {
    if (ncalls == 0)
    {
      printf("%5s %5s %16s %16s %16s %16s\n",
        "PASS","ITER","POBJ","PINFEAS","DINFEAS","BESTOBJ");
    }
    LSgetCallbackInfo(model,iLoc,LS_IINFO_NLP_ITER,&iter);
    LSgetCallbackInfo(model,iLoc,LS_DINFO_POBJ,&pobj);
    LSgetCallbackInfo(model,iLoc,LS_DINFO_PINFEAS,&pfeas);
    LSgetCallbackInfo(model,iLoc,LS_DINFO_DINFEAS,&dfeas);
    printf("%5s %5d %16.5e %16.5e %16.5e %16s\n",
      "",iter,pobj,pfeas,dfeas,"");
  }
  ncalls++;
  return 0;
} /*print_log*/
/*************************************************************
   Callback function to compute function values
 *************************************************************/
int    CALLBACKTYPE Funcalc8(pLSmodel pModel,void    *pUserData,
                             int      nRow  ,double  *pdX,
                             int      nJDiff,double  dXJBase,
                             double   *pdFuncVal,int  *pReserved)
{
  double val=0.0, X = pdX[0], Y = pdX[1];
  int    nerr=0;
  /* compute objective's functional value*/
```

```
  if (nRow==-1)
    val = 3*f1(X,Y)*g1(X,Y) - 10*f2(X,Y)*g2(X,Y) - g3(X,Y)/3;
  /* compute constraint 0's functional value */
  else if (nRow==0)
    val = X*X + Y - 6.0;
  /* compute constraint 1's functional value */
  else if (nRow==1)
    val = X + Y*Y - 6.0;
  *pdFuncVal=val;
  return nerr;
} /*Funcalc8*/
/*********************************************************
  Callback function to compute derivatives
 *********************************************************/
int CALLBACKTYPE Gradcalc8(pLSmodel pModel, void *pUserData,
                           int nRow,double *pdX, double *lb,
                           double *ub, int nNewPnt, int nNPar,
                           int *parlist, double *partial)
{
  int i2,nerr=0;
  double X=pdX[0], Y=pdX[1];
  /*zero out the partials */
  for (i2=0;i2<nNPar;i2++) partial[i2]=0.0;
  /* partial derivatives of the objective function */
  if (nRow==-1) {
     for (i2=0;i2<nNPar;i2++) {
       if (lb[parlist[i2]]!=ub[parlist[i2]]) {
           if (parlist[i2]==0) {
             partial[i2]=
                  3*(dxf1(X,Y)*g1(X,Y) + f1(X,Y)*dxg1(X,Y) )
              -  10*(dxf2(X,Y)*g2(X,Y) + f2(X,Y)*dxg2(X,Y) )
              - 1/3*(dxg3(X,Y));
           } else if (parlist[i2]==1) {
             partial[i2]=
                  3*(dyf1(X,Y)*g1(X,Y) + f1(X,Y)*dyg1(X,Y) )
              -  10*(dyf2(X,Y)*g2(X,Y) + f2(X,Y)*dyg2(X,Y) )
              - 1/3*(dyg3(X,Y));
           }
       }
     }
  }
  /* partial derivatives of Constraint 0 */
  else if (nRow==0) {
     for (i2=0;i2<nNPar;i2++) {
       if (lb[parlist[i2]]!=ub[parlist[i2]]) {
         if (parlist[i2]==0) {
           partial[i2]=2.0*X;
         } else if (parlist[i2]==1) {
           partial[i2]=1;
         }
       }
     }
  }
  /* partial derivatives of Constraint 1 */
  else if (nRow==1) {
     for (i2=0;i2<nNPar;i2++) {
```

```
        if (lb[parlist[i2]]!=ub[parlist[i2]]) {
          if (parlist[i2]==0) {
            partial[i2]=1;
          } else if (parlist[i2]==1) {
            partial[i2]=2.0*Y;
          }
        }
      }
    }
    return nerr;
}
/* main entry point*/
int main(int argc, char **argv)
{
    pLSenv env      = NULL;
    pLSmodel model  = NULL;
    FILE *logfile   = stdout;
    int errors=0,errorcode=LSERR_NO_ERROR, status;
    double lb[2],ub[2],A[4],rhs[2],cost[2], primal[2],objval;
    int Abegcol[3],Arowndx[4],Alencol[2],Nobjndx[2];
    int m,n,nz,  Nnlobj, counter = 0;
    char contype[2];
    char MY_LICENSE_KEY[1024];
   /*********************************************************
    * Step 1: Create a model in the environment.
    *********************************************************/
    errorcode = LSloadLicenseString(
                "../../../license/lndapi60.lic",MY_LICENSE_KEY);
    env = LScreateEnv(&errorcode,MY_LICENSE_KEY);
    if (errorcode!=LSERR_NO_ERROR) return errorcode;
    model = LScreateModel(env,&errorcode);
    if (errorcode!=LSERR_NO_ERROR) return errorcode;
   /*********************************************************
    * Step 2: Specify the LP portion of the model.
    *********************************************************/
    /* model dimensions */
    m = n = 2;  nz = 4;
    /* The indices of the first nonzero in each column */
    Abegcol[0]=0; Abegcol[1]=2; Abegcol[2]=4;
    /* The length of each column */
    Alencol[0]=2; Alencol[1]=2;
    /* The row indices of the nonzero coefficients */
    Arowndx[0]=0; Arowndx[1]=1;  Arowndx[2]=0;  Arowndx[3]=1;
    /* The nonzero coefficients of the linear portion of the model*/
    /* The NLP elements have a zero at each of their occurrence    */
    A[0]=0.0; A[1]=1.0; A[2]=1.0; A[3]=0.0;
    /* The objective coefficients of the linear portion of the model*/
    cost[0]=0.0; cost[1]=0.0;
    /* lower bounds on variables */
    lb[0]=-3.0  ; ub[0]= 3.0;   lb[1]=-3.0  ; ub[1]= 3.0;
    /* The right-hand sides of the constraints */
    rhs[0]=0.0; rhs[1]=0.0;
    /* The constraint types */
    contype[0]='L'; contype[1]='L';
    /* Load in nonzero structure and linear/constant terms.  */
    errorcode=LSloadLPData(model,m,n,LS_MIN,0.0,cost,rhs,contype,nz,
```

```
                          Abegcol,Alencol,A,Arowndx,lb,ub);
  if (errorcode!=LSERR_NO_ERROR) return errorcode;
 /*********************************************************
  * Step 3: Specify the NLP portion of the model.
  *********************************************************/
  /* The number of nonlinear variables in each column */
  Alencol[0]=1; Alencol[1]=1;
  /* The indices of the first nonlinear variable in each column */
  Abegcol[0]=0; Abegcol[1]=1; Abegcol[2]=2;
  /* The indices of nonlinear constraints */
  Arowndx[0]=0;
  Arowndx[1]=1;
  /* The indices of variables that are nonlinear in the objective*/
  Nobjndx[0]=0;
  Nobjndx[1]=1;
  /* Number nonlinear variables in cost. */
  Nnlobj = 2;
  /* Load the nonlinear structure */
  errorcode=LSloadNLPData(model,Abegcol,Alencol,
            NULL,Arowndx,Nnlobj,Nobjndx,0);
  printf("\nThe model is installed successfully...\n");
 /*********************************************************
  * Step 4: Set up callback functions
  *********************************************************/
  /* Install the routine that will calculate the function values. */
  errorcode=LSsetFuncalc(model,(Funcalc_type) Funcalc8,NULL);
  if (errorcode!=LSERR_NO_ERROR) return errorcode;
  /* Install the routine that will calculate the gradient */
  errorcode=LSsetGradcalc(model,Gradcalc8,NULL,0,NULL);
  if (errorcode!=LSERR_NO_ERROR) return errorcode;
  /* Install a callback function */
  LSsetCallback(model,(cbFunc_t) print_log, &counter);

    /* Set the print level to 1 */
  errorcode=LSsetModelIntParameter(model,LS_IPARAM_NLP_PRINTLEVEL,1);
  /* Turn multi-start search on */
LSsetModelIntParameter(model,LS_IPARAM_NLP_SOLVER,LS_NMETHOD_MSW_GRG)
;
  /* Set maximum number of local optimizations */
  LSsetModelIntParameter(model,LS_IPARAM_NLP_MAXLOCALSEARCH,1);
 /*********************************************************
  * Step 5: Solve the model
  *********************************************************/
  /* load an initial starting point */
  primal[0] = 0.25;  primal[1] = -1.65;
  errorcode=LSloadVarStartPoint(model, primal);
  /* optimize the model */
  errorcode=LSoptimize(model,LS_METHOD_FREE, &status);
  if (errorcode!=LSERR_NO_ERROR)
    return errorcode;
  {
    int i;
    errorcode = LSgetInfo(model, LS_DINFO_POBJ, &objval);
    errorcode = LSgetPrimalSolution(model, primal);
    printf("\n\n\nPrinting the best local optimum found.\n");
    printf("obj  = %f \n",objval);
```

```
    for (i=0; i<2; i++) printf("x[%d] = %f \n",i,primal[i]);
  }
 /*********************************************************
  * Step 6: Delete the model & env space
  *********************************************************/
  LSdeleteModel(&model);
  LSdeleteEnv(&env);
  /* Wait until user presses the Enter key */
   printf("Press <Enter> ...");
   getchar();
  return errorcode;
} /*main*/
```

The source code file for this example may be found in the *\lindoapi\samples\c\ex_nlp1* folder. After creating the executable "ex_nlp1.exe", you can run the application from the DOS-prompt and produce the following report on the screen.

```
The model is installed successfully...
PASS     ITER     POBJ            PINFEAS          DINFEAS       BESTOBJ
         0     0.00000e+000     0.00000e+000     0.00000e+000
         1     0.00000e+000     0.00000e+000     0.00000e+000
         2     0.00000e+000     0.00000e+000     0.00000e+000
         3    -6.54423e+000     0.00000e+000     6.15217e+000
         4    -6.54480e+000     0.00000e+000     5.97951e+000
         5     2.26638e-003     0.00000e+000     5.90105e+000
         6    -7.50481e-003     0.00000e+000     1.59402e-001
         6    -7.50481e-003     0.00000e+000     1.59402e-001
         6     0.00000e+000     0.00000e+000     0.00000e+000
         7     0.00000e+000     0.00000e+000     0.00000e+000
         8     0.00000e+000     0.00000e+000     0.00000e+000
         9    -7.50509e-003     0.00000e+000     4.32958e-001
        10    -3.81927e-001     0.00000e+000     4.32968e-001
        11    -4.28345e-001     0.00000e+000     2.43317e+000
        12    -4.86107e-001     0.00000e+000     1.98075e+000
        13    -1.22076e+000     0.00000e+000     3.24088e+000
        14    -1.46611e+000     0.00000e+000     1.34246e+001
        15    -2.45416e+000     0.00000e+000     2.11428e+001
        16    -2.85036e+000     0.00000e+000     7.38464e+000
        17    -3.01813e+000     0.00000e+000     1.31130e+001
        18    -3.01813e+000     0.00000e+000     1.17374e+000
        19    -2.97944e+000     0.00000e+000     1.17374e+000
        19    -2.97944e+000     0.00000e+000     1.17374e+000
        19    -2.97944e+000     0.00000e+000     1.17374e+000

Printing the best local optimum found.
obj  = -2.979441
x[0] = -1.449174
x[1] = 0.194467
Press <Enter> ...
```

Example 2: Instruction-List Style Interface

This example illustrates the use of LINDO API to build and solve a small nonlinear mixed integer model loaded via the instruction-list interface.

```
/*
################################################################
#                        LINDO-API
#                     Sample Programs
#                  Copyright (c) 2007 by LINDO Systems, Inc
#
#         LINDO Systems, Inc.           312.988.7422
#         1415 North Dayton St.         info@lindo.com
#         Chicago, IL 60622             http://www.lindo.com
################################################################
  File   : ex_nlp2.c
  Purpose: Solve a NLP using the instruction-list style interface.
  Model  : A nonlinear model with multiple local minimizers.

            maximize  abs( x0 + 1) + .4 * x1;
            s.t.     x0           + x1 - 4      <= 0;
                     x0 * x1      + x1 - 6      <= 0;
                     x0 * x1                    <= 0;
                     max(x0 , x1 + 1)           >= 0;
                     if(x1, 1, x1)              <= 0;
                     (x1 * 2 * x1  -  x1) * x0  <= 0;
                     -100  <=  x0  <=  100
                     x1 is binary
*/
#include <stdio.h>
#include <stdlib.h>
/* LINDO API header file */
#include "lindo.h"
/* Define a macro to declare variables for
    error checking */
#define APIERRORSETUP
   int nErrorCode;
   char cErrorMessage[LS_MAX_ERROR_MESSAGE_LENGTH]
/* Define a macro to do our error checking */
#define APIERRORCHECK
   if (nErrorCode)
   {
      if ( pEnv)
      {
         LSgetErrorMessage( pEnv, nErrorCode,
          cErrorMessage);
         printf("nErrorCode=%d:  %s\n", nErrorCode,
          cErrorMessage);
      } else {
         printf( "Fatal Error\n");
      }
      exit(1);
   }
/* main entry point */
int main()
{
```

```
   APIERRORSETUP;
   pLSenv pEnv;
   pLSmodel pModel;
   char MY_LICENSE_KEY[1024];
 /*********************************************************
  * Step 1: Create a model in the environment.
  *********************************************************/
   nErrorCode = LSloadLicenseString(
                "../../../license/lndapi60.lic",MY_LICENSE_KEY);

   pEnv = LScreateEnv ( &nErrorCode, MY_LICENSE_KEY);
   if ( nErrorCode == LSERR_NO_VALID_LICENSE)
   {
      printf( "Invalid License Key!\n");
      exit( 1);
   }
   APIERRORCHECK;
/* >>> Step 2 <<< Create a model in the environment. */
   pModel = LScreateModel(pEnv,&nErrorCode);
   APIERRORCHECK;
   {
/* >>>> Step 3 <<< Set up the instruction list of the model. */
      int nobjs, ncons, nvars, nnums, lsize;
      int objsense[1];
      char ctype[6], vtype[2];
      int code[100], varndx[2];
      double numval[8],varval[2];
      int objs_beg[1], objs_length[1], cons_beg[6], cons_length[6];
      double lwrbnd[2], uprbnd[2];
      int nLinearz, nAutoDeriv;
      int ikod, iobj, icon;
      /* Number of constraints */
      ncons = 6;
      /* Number of objectives */
      nobjs = 1;
      /* Number of variables */
      nvars = 2;
      /* Number of real number constants */
      nnums = 5;
      /* Variable index */
      varndx[0]=1;
      varndx[1]=2;
      /* Lower bounds of variables */
      lwrbnd[0]=-100.0;
      lwrbnd[1]=0.0;
      /* Upper bounds of variables */
      uprbnd[0]=100.0;
      uprbnd[1]=1.0;
      /* Starting point of variables */
      varval[0]=4.0;
      varval[1]=0.0;
      /* Variable type, C= continuous, B = binary */
      vtype[0] = 'C';
      vtype[1] = 'B';
       /* Double Precision constants in the model */
      numval[0]=1.0;
```

```
numval[1]=0.4;
numval[2]=6.0;
numval[3]=4.0;
numval[4]=2.0;
/* Count for instruction code */
   ikod = 0;
/* Count for objective row */
   iobj = 0;
/* Count for constraint row */
   icon = 0;
/*
 *  Instruction code of the objective:
 *
 *  max abs( x0 + 1) + .4 * x1;
 */
/* Direction of optimization */
objsense[iobj]= LS_MAX;
/* Beginning position of objective */
objs_beg[iobj]=ikod;
/* Instruction list code */
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    0;
code[ikod++]=  EP_PUSH_NUM;
code[ikod++]=    0;
code[ikod++]=  EP_PLUS;
code[ikod++]=  EP_ABS;
code[ikod++]=  EP_PUSH_NUM;
code[ikod++]=    1;
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    1;
code[ikod++]= EP_MULTIPLY;
code[ikod++]= EP_PLUS;
 /* Length of objective */
objs_length[iobj] = ikod - objs_beg[iobj];

/*
 *  Instruction code of constraint 0:
 *
 *  x0  + x1 - 4 <= 0;
 */
 /* Constraint type */
ctype[icon]= 'L';   /* less or than or equal to */
/* Beginning position of constraint 0 */
cons_beg[icon]= ikod;
/* Instruction list code */
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    0;
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    1;
code[ikod++]=  EP_PLUS;
code[ikod++]=  EP_PUSH_NUM;
code[ikod++]=    3;
code[ikod++]=  EP_MINUS;
 /* Length of constraint 0 */
cons_length[icon] = ikod - cons_beg[icon];
/* Increment the constraint count */
```

```
 icon++;
/*
 *  Instruction code of constraint 1:
 *
 *  x0 * x1      + x1 - 6 <= 0;
 */
 /* Constraint type */
ctype[icon]= 'L';   /* less than or equal to */
/* Beginning position of constraint 1 */
cons_beg[icon]=  ikod;
/* Instruction list code */
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    0;
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    1;
code[ikod++]=  EP_MULTIPLY;
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    1;
code[ikod++]=  EP_PLUS;
code[ikod++]=  EP_PUSH_NUM;
code[ikod++]=    2;
code[ikod++]=  EP_MINUS;
/* Length of constraint 1 */
cons_length[icon] = ikod - cons_beg[icon];
/* Increment the constraint count */
 icon++;
/*
 *  Instruction code of constraint 2:
 *
 *  x0 * x1           <= 0;
 */
 /* Constraint type */
ctype[icon]= 'L';   /* less than or equal to */
/* Beginning position of constraint 2 */
cons_beg[icon]=  ikod;
/* Instruction list code */
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    0;
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    1;
code[ikod++]=  EP_MULTIPLY;
 /* Length of constraint 2 */
cons_length[icon] = ikod - cons_beg[icon];
/* Increment the constraint count */
icon++;
/*
 *  Instruction code of constraint 3:
 *
 *  max(x0 , x1 + 1)        >= 0;
 */
 /* Constraint type */
ctype[icon]= 'G';   /* greater than or equal to */
/* Beginning position of constraint 3 */
cons_beg[icon]=  ikod;
/* Instruction list code */
code[ikod++]=  EP_PUSH_VAR;
```

```
code[ikod++]=    0;
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    1;
code[ikod++]=  EP_PUSH_NUM;
code[ikod++]=    0;
code[ikod++]=  EP_PLUS;
code[ikod++]=  EP_MAX;
code[ikod++]=    2;
 /* Length of constraint 3 */
cons_length[icon] = ikod - cons_beg[icon];
/* Increment the constraint count */
 icon++;
/*
 *  Instruction code of constraint 4:
 *
 *  if(x1, 1, x1)        <= 0;
 */
 /* Constraint type */
ctype[icon]= 'L';  /* less than or equal to */
/* Beginning position of constraint 4 */
cons_beg[icon]=  ikod;
/* Instruction list code */
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    1;
code[ikod++]=  EP_PUSH_NUM;
code[ikod++]=    0;
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    1;
code[ikod++]=  EP_IF;
 /* Length of constraint 4 */
cons_length[icon] = ikod - cons_beg[icon];
/* Increment the constraint count */
 icon++;
/*
 *  Instruction code of constraint 5:
 *
 *  (x1 * 2 * x1  -  x1) * x0      <= 0;
 */
 /* Constraint type */
ctype[icon]= 'L';  /* less than or equal to */
/* Beginning position of constraint 5 */
cons_beg[icon]=  ikod;
/* Instruction list code */
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    1;
code[ikod++]=  EP_PUSH_NUM;
code[ikod++]=    4;
code[ikod++]=  EP_MULTIPLY;
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    1;
code[ikod++]=  EP_MULTIPLY;
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    1;
code[ikod++]=  EP_MINUS;
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    0;
```

```
        code[ikod++]=  EP_MULTIPLY;
         /* Length of constraint 5 */
        cons_length[icon] = ikod - cons_beg[icon];

        /* Total number of items in the instruction list */
        lsize = ikod;
        /* Set linearization level, before a call to LSloadInstruct.
         * If not specified, the solver will decide */
        nLinearz = 1;
        nErrorCode = LSsetModelIntParameter (pModel,
                      LS_IPARAM_NLP_LINEARZ, nLinearz);
        APIERRORCHECK;

        /* Set up automatic differentiation, before a call to
         * LSloadInstruct. If not specified, the numerical derivative
         * will be applied */
        nAutoDeriv = 1;
        nErrorCode = LSsetModelIntParameter (pModel,
                      LS_IPARAM_NLP_AUTODERIV, nAutoDeriv);
        APIERRORCHECK;
        /* Pass the instruction list to problem structure
         * by a call to LSloadInstruct() */
        nErrorCode = LSloadInstruct (pModel, ncons, nobjs, nvars,
nnums,
                      objsense, ctype,  vtype, code, lsize, varndx,
                      numval, varval, objs_beg, objs_length, cons_beg,
                      cons_length, lwrbnd, uprbnd);
        APIERRORCHECK;
    }
/*
 * >>> Step 5 <<< Perform the optimization using the MIP solver
 */
    nErrorCode = LSsolveMIP(pModel, NULL);
    APIERRORCHECK;
    {
        int nLinearity;
        double objval=0.0, primal[100];
        /* Get the optimization result */
        LSgetInfo(pModel, LS_DINFO_MIP_OBJ, &objval);
        APIERRORCHECK;
        LSgetMIPPrimalSolution( pModel, primal) ;
        APIERRORCHECK;
        printf("\n\nObjective = %f \n",objval);
        printf("x[0] = %f \n",primal[0]);
        printf("x[1] = %f \n",primal[1]);
        /* Get the linearity of the solved model */
        nErrorCode = LSgetModelIntParameter (pModel,
                     LS_IPARAM_NLP_LINEARITY, &nLinearity);
        APIERRORCHECK;
        /* Report the status of solution */
        if (nLinearity)
        printf("\nModel has been completely linearized.\
                \nSolution Status: Globally Optimal\n");
        else
        printf("\nModel is nonlinear.\
                \nSolution Status: Locally Optimal\n\n");
```

```
    }
  /* >>> Step 7 <<< Delete the LINDO environment */
    LSdeleteEnv(&pEnv);
   /* Wait until user presses the Enter key */
    printf("Press <Enter> ...");
    getchar();
}
```

This example maximizes a nonlinear objective subject to one linear and five nonlinear constraints. After including the C header file, creating a LINDO environment object, and creating a model object within the environment, the model is then assembled via the instruction list code. First, the numbers of objective, constraints, variables, and constants that appeared in the model are set with the following:

```
/* Number of objectives */
nobjs = 1;
/* Number of constraints */
ncons = 6;
/* Number of variables */
nvars = 2;
/* Number of real number constants */
nnums = 5;
```

Then, all variable related information is defined, including lower bounds, upper bounds, variable types, starting points, and variable indices. The setting of starting points and variable indices are optional to the user.

```
/* Lower bounds of variables */
lwrbnd[0]=-100.0;
lwrbnd[1]=0.0;
/* Upper bounds of variables */
uprbnd[0]=100.0;
uprbnd[1]=1.0;
/* Starting point of variables */
varval[0]=4.0;
varval[1]=0.0;
/* Variable type, C= continuous, B = binary */
vtype[0] = 'C';
vtype[1] = 'B';
/* Variable index */
varndx[0]=1;
varndx[1]=2;
```

Next, all double precision constants used in the model are placed into a number array:

```
/* Double Precision constants in the model */
numval[0]=1.0;
numval[1]=0.4;
numval[2]=6.0;
numval[3]=4.0;
numval[4]=2.0;
```

Right before starting to build up instruction lists, the counts for instruction codes, objective rows, and constraint rows are reset with the following:

```
/* Count for instruction code */
  ikod = 0;
/* Count for objective row */
  iobj = 0;
/* Count for constraint row */
  icon = 0;
```

The instruction lists and related information are then constructed row by row. The objective function in our example is to maximize a nonlinear function involving the *abs*() function:

```
/*
 *  Instruction code of the objective:
 *
 *  max abs( x0 + 1) + .4 * x1;
 */
```

For the objective row, the type of row is defined first by setting the direction of this objective:

```
/* Direction of optimization */
objsense[iobj]= LS_MAX;
```

The beginning position of the objective in the instruction list vector is set at the current count on the instruction code:

```
/* Beginning position of objective */
objs_beg[iobj]=ikod;
```

Following the principles of postfix, the corresponding instruction list of the objective function is placed into the code vector accordingly:

```
/* Instruction list code */
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    0;
code[ikod++]=  EP_PUSH_NUM;
code[ikod++]=    0;
code[ikod++]=  EP_PLUS;
code[ikod++]=  EP_ABS;
code[ikod++]=  EP_PUSH_NUM;
code[ikod++]=    1;
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    1;
code[ikod++]= EP_MULTIPLY;
code[ikod++]= EP_PLUS;
```

The length of the objective is calculated as the difference of the current count of codes and the beginning position:

```
/* Length of objective */
objs_length[iobj] = ikod - objs_beg[iobj];
```

Since there is only a single objective, the constraint rows are developed. The first constraint row, constraint 0, is a simple linear equation constrained to less-than-or-equal-to zero:

```
/*
 *  Instruction code of constraint 0:
 *
 *  x0  + x1 - 4 <= 0;
 */
```

For this constraint, the type of constraint must first be defined to be less-than-or-equal-to:

```
/* Constraint type */
ctype[icon]= 'L';   /* less or than or equal to */
```

The beginning position of the constraint in the instruction list vector is set at the current count on the instruction code:

```
/* Beginning position of constraint 0 */
cons_beg[icon]= ikod;
```

Again, following the principles of postfix, the corresponding instruction list of this constraint function is placed into the code vector accordingly:

```
/* Instruction list code */
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    0;
code[ikod++]=  EP_PUSH_VAR;
code[ikod++]=    1;
code[ikod++]=  EP_PLUS;
code[ikod++]=  EP_PUSH_NUM;
code[ikod++]=    3;
code[ikod++]=  EP_MINUS;
```

At the end, the length of the current instruction list is set:

```
/* Length of constraint 0 */
cons_length[icon] = ikod - cons_beg[icon];
```

The count on constraint rows is then incremented by one:

```
/* Increment the constraint count */
icon++;
```

Following the same rule, the instruction lists for constraint 1, 2, 3, 4, and 5 can also be built accordingly. After completely specifying the instruction lists and their related information, this model definition segment is finished by declaring the total number of codes in the instruction lists:

```
/* Total number of items in the instruction list */
lsize = ikod;
```

LINDO API provides an user option in dealing with the model, which is linearization. To use this option, it should be specified before you call *LSloadInstruct* to load nonlinear codes. The example model contains nonlinear components of *abs*(), *if*(), complementary constraint, and $x^* y$ (where x and/or y are binary 0/1 variables). All of these nonlinear components are linearizable. Therefore, if the

Maximum linearization option is selected, the model can be completely linearized when loaded into LINDO API. In such a case, the model will be transformed into an equivalent linear format, which need not set up the differentiation option.

Note:	Constraint 5 involves a multiplication of a parenthetical expression (x1*2*x1 - x1) with variable x0, which is expanded into x1*2*x1*x0 –x1*x0 and linearized accordingly.

On the other hand, if the *None* linearization option is selected and the model stays in its nonlinear form when loaded into LINDO API, using automatic differentiation can help the solver converge to the optimal solution in a faster and more precise manner. Otherwise, the solver will use the default, finite difference differentiation. In this example, the linearization option is turned off and differentiation is set to automatic with the following code segment:

```
/* Set linearization level, before a call to LSloadInstruct.
 * If not specified, the solver will decide */
nLinearz = 1;
nErrorCode = LSsetModelIntParameter (pModel,
              LS_IPARAM_NLP_LINEARZ, nLinearz);
APIERRORCHECK;
/* Set up automatic differentiation. If not specified, the numerical
   derivative will be applied */
nAutoDeriv = 1;
nErrorCode = LSsetModelIntParameter (pModel,
              LS_IPARAM_NLP_AUTODERIV, nAutoDeriv);
APIERRORCHECK;
```

The next step, step 5, is to perform the optimization of the model with a call to *LSsolveMIP* and retrieve the variable values. For a more detailed description of this step, please refer to the previous chapters. LINDO API also provides a parameter *LS_IPARAM_NLP_LINEARITY* for the user to check the characteristic of the solved model:

```
/* Get the linearity of the solved model */
nErrorCode = LSgetModelIntParameter (pModel,
             LS_IPARAM_NLP_LINEARITY, &nLinearity);
APIERRORCHECK;
```

If the returning value of linearity equals one, then the model is linear or has been completely linearized in the linearization step. Thus, the global optimality of solution can be ascertained.

The source code file for this example may be found in the \lindoapi\samples\c\ex_nlp2\ folder. After creating the executable “ex_nlp2.exe”, the application can be run from the DOS-prompt and produce the following report on your screen.

```
Objective = 5.000000
x[0] = 4.000000
x[1] = 0.000000

Model is nonlinear.
Solution Status: Locally Optimal

Press <Enter> ...
```

Example 3: Multistart Solver for Non-Convex Models

This example demonstrates how the multistart nonlinear solver can be used in solving a non-convex mixed-integer nonlinear program. The example uses the same model given in Example 1 with the black-box style interface where gradients are computed using finite differences. A callback function is included, so each local solution found during the solution procedure is reported to the user. For more information on callback functions, refer to Chapter 9, *Using Callback Functions*.

```
/*
#################################################################
#                       LINDO-API
#                    Sample Programs
#                  Copyright (c) 2007 by LINDO Systems, Inc
#
#         LINDO Systems, Inc.           312.988.7422
#         1415 North Dayton St.         info@lindo.com
#         Chicago, IL 60622             http://www.lindo.com
#################################################################
  File   : ex_nlp3.c
  Purpose: Solve a MINLP using the black-box stye interface.
  Model  : A nonlinear model with linear constraints.
        minimize  f(x,y) =  3*(1-x)^2*exp(-(x^2) - (y+1)^2)
                          - 10*(x/5 - x^3 - y^5).*exp(-x^2-y^2)
                          - 1/3*exp(-(x+1)^2 - y^2);
        subject to
                          x  + y   <=  3;
                             - y   <=  1;
                          x  integer
*/
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "lindo.h"

/* the summands of the objective function */
#define  g1(X,Y) ( exp( -pow(X  ,2)  - pow(Y+1,2) )  )
#define  g2(X,Y) ( exp( -pow(X  ,2)  - pow(Y  ,2) )  )
#define  g3(X,Y) ( exp( -pow(X+1,2)  - pow(Y  ,2) )  )
#define  f1(X,Y) (        pow(1-X,2)                 )
#define  f2(X,Y) ( X/5 - pow(X  ,3)  - pow(Y  ,5)    )
/*************************************************************
   Standard callback function to display local solutions
 *************************************************************/
int  CALLTYPE local_sol_log(pLSmodel model,int iLoc, void *cbData)
{
  int iter=0,niter,biter,siter;
  int *nKKT = (int *) cbData, npass, nbrn;
  double pfeas=0.0,pobj=0.0;
  double bestobj;
  if (iLoc==LSLOC_LOCAL_OPT)
  {
    if (*nKKT == 0){
      printf(" %5s %11s %11s %11s %10s\n",
        "Iter","Objective","Infeas","Best","Branches");
    }
```

```
    LSgetCallbackInfo(model,iLoc,LS_IINFO_MIP_NLP_ITER,&niter);
    LSgetCallbackInfo(model,iLoc,LS_IINFO_MIP_SIM_ITER,&siter);
    LSgetCallbackInfo(model,iLoc,LS_IINFO_MIP_BAR_ITER,&biter);
    LSgetCallbackInfo(model,iLoc,LS_DINFO_POBJ,&pobj);
    LSgetCallbackInfo(model,iLoc,LS_DINFO_PINFEAS,&pfeas);
    LSgetCallbackInfo(model,iLoc,LS_DINFO_MSW_POBJ,&bestobj);
    LSgetCallbackInfo(model,iLoc,LS_IINFO_MIP_BRANCHCOUNT,&nbrn);
    iter = niter+siter+biter;
    printf(" %5d %11.3f %11.3f %11.3f %10d\n",iter,pobj,pfeas,
      bestobj,nbrn);
    (*nKKT)++;
  }
  return 0;
} /*local_sol_log*/
/*********************************************************
   Callback function to compute function values
 *********************************************************/
int    CALLBACKTYPE Funcalc8(pLSmodel pModel,void    *pUserData,
                             int      nRow  ,double  *pdX,
                             int      nJDiff,double  dXJBase,
                             double   *pdFuncVal,int  *pReserved)
{
  double val=0.0, X = pdX[0], Y = pdX[1];
  int    nerr=0;
  /* compute objective's functional value*/
  if (nRow==-1)
    val = 3*f1(X,Y)*g1(X,Y) - 10*f2(X,Y)*g2(X,Y) - g3(X,Y)/3;
  /* compute constaint 0's functional value */
  else if (nRow==0)
    val = X + Y - 3.0;
  /* compute constaint 1's functional value */
  else if (nRow==1)
    val = - Y - 1.0;
  *pdFuncVal=val;
  return nerr;
} /*Funcalc8*/

/* main entry point*/
int main(int argc, char **argv)
{
  pLSenv env      = NULL;
  pLSmodel model  = NULL;
  FILE *logfile   = stdout;
  int errors=0,errorcode=LSERR_NO_ERROR;
  double lb[2],ub[2],A[4],rhs[2],cost[2];
  int Abegcol[3],Arowndx[4],Alencol[2],Nobjndx[2];
  int m,n,nz, Nnlobj, howmany=0;
  char contype[2],vartype[2];
  char MY_LICENSE_KEY[1024];
 /*********************************************************
  * Step 1: Create a model in the environment.
  *********************************************************/
  errorcode = LSloadLicenseString(
              "../../../license/lndapi60.lic",MY_LICENSE_KEY);

  env = LScreateEnv(&errorcode,MY_LICENSE_KEY);
```

```
 if (errorcode!=LSERR_NO_ERROR) return errorcode;
 model = LScreateModel(env,&errorcode);
 if (errorcode!=LSERR_NO_ERROR) return errorcode;
/*********************************************************
 * Step 2: Specify the LP portion of the model.
 *********************************************************/
 /* model dimensions */
 m = n = 2;  nz = 3;
 /* The indices of the first nonzero in each column */
 Abegcol[0]=0; Abegcol[1]=1; Abegcol[2]=3;
 /* The length of each column */
 Alencol[0]=1; Alencol[1]=2;
 /* The row indices of the nonzero coefficients */
 Arowndx[0]=0; Arowndx[1]=0;  Arowndx[2]=1;
 /* The nonzero coefficients of the linear portion of the model*/
 /* The NLP elements have a zero at each of their occurence    */
 A[0]=1.0; A[1]=1.0; A[2]=-1.0;
 /* The objective coefficients of the linear portion of the model*/
 cost[0]=0.0; cost[1]=0.0;
 /* lower bounds on variables */
 lb[0]=-3.0  ; ub[0]= 3.0;   lb[1]=-3.0  ; ub[1]= 3.0;
 /* The right-hand sides of the constraints */
 rhs[0]=3.0; rhs[1]=1.0;
 /* The constraint types */
 contype[0]='L'; contype[1]='L';
 vartype[0]='I'; vartype[1]='C';
 /* Load in nonzero structure and linear/constant terms.  */
 errorcode=LSloadLPData(model,m,n,LS_MIN,0.0,cost,rhs,contype,nz,
                        Abegcol,Alencol,A,Arowndx,lb,ub);
 if (errorcode!=LSERR_NO_ERROR) return errorcode;
 errorcode=LSloadVarType(model,vartype);
 if (errorcode!=LSERR_NO_ERROR) return errorcode;
 errorcode=LSwriteLINDOFile(model,"lpModel.ltx");
 if (errorcode!=LSERR_NO_ERROR) return errorcode;
/*********************************************************
 * Step 3: Specify the NLP portion of the model.
 *********************************************************/
 /* The number of nonlinear variables in each column */
 Alencol[0]=0; Alencol[1]=0;

 /* The indices of the first nonlinear variable in each column */
 Abegcol[0]=0; Abegcol[1]=0; Abegcol[2]=0;

 /* The indices of nonlinear constraints */
 Arowndx[0]=0;

 /* The indices of variables that are nonlinear in the objective*/
 Nobjndx[0]=0;  Nobjndx[1]=1;

 /* Number nonlinear variables in cost. */
 Nnlobj = 2;
 /* Load the nonlinear structure */
 errorcode=LSloadNLPData(model,Abegcol,Alencol,
           NULL,Arowndx,Nnlobj,Nobjndx,NULL);

 if (errorcode!=LSERR_NO_ERROR) return errorcode;
```

```
  /*********************************************************
   * Step 4: Set up callback functions
   *********************************************************/
   /* Install the callback function to call at every local solution */
   LSsetCallback(model,(cbFunc_t) local_sol_log,&howmany);

   /* Set the print level to 1 */
   errorcode=LSsetModelIntParameter(model,LS_IPARAM_NLP_PRINTLEVEL,1);

   /* Set the NLP prelevel to 126 */
   errorcode=LSsetModelIntParameter(model,LS_IPARAM_NLP_PRELEVEL,0);

   /* Install the routine that will calculate the function values. */
   errorcode=LSsetFuncalc(model,(Funcalc_type) Funcalc8,NULL);
   if (errorcode!=LSERR_NO_ERROR) return errorcode;
   /*********************************************************
   * Step 5: Solve the model
   *********************************************************/
   /* Turn multi-start search on */
LSsetModelIntParameter(model,LS_IPARAM_NLP_SOLVER,LS_NMETHOD_MSW_GRG)
;
   /* Set maximum number of local optimizations */
   LSsetModelIntParameter(model,LS_IPARAM_NLP_MAXLOCALSEARCH,5);

   printf("\n\tSolving the MINLP using Multi-Start Approach.\n\n");
   errorcode=LSsolveMIP(model,NULL);
   if (errorcode!=LSERR_NO_ERROR) return errorcode;
   {
     int i;
     double objval, primal[2];
     errorcode = LSgetMIPPrimalSolution(model, primal);
     errorcode = LSgetInfo(model, LS_DINFO_MIP_OBJ, &objval);
     if (errorcode == LSERR_NO_ERROR)
     {
       printf("\n\n\n");
       printf("obj  = %15.7f \n",objval);
       for (i=0; i<2; i++) printf("x[%d] = %15.7f \n",i,primal[i]);
     }
     else
     {
       printf("Error %d occured\n\n\n",errorcode);
     }
   }
  /*********************************************************
   * Step 6: Delete the model & env space
   *********************************************************/
   LSdeleteModel(&model);
   LSdeleteEnv(&env);

   /* Wait until user presses the Enter key */
    printf("Press <Enter> ...");
    getchar();

   return errorcode;
} /*main*/
```

The source code file for this example may be found in the \lindoapi\samples\c\ex_nlp3\ folder. After creating the executable "ex_nlp3.exe", the application can be run from the DOS-prompt and produce the following report on your screen.

```
          Solving the MINLP using Multi-Start Approach.

  Iter   Objective      Infeas        Best   Branches
    10      -0.032       0.000      -0.032          0
    17       0.013       0.000      -0.032          0
    33      -0.032       0.000      -0.032          0
    40       0.013       0.000      -0.032          0
    74      -0.032       0.000      -0.032          0
    81       0.013       0.000      -0.032          0
   106      -0.032       0.000      -0.032          1
   113       0.013       0.000      -0.032          1
   138      -0.009       0.000      -0.009          2
   142       0.013       0.000       0.013          3


obj  =      -0.0087619
x[0] =      -3.0000000
x[1] =      -1.0000000
Press <Enter> ...
```

As seen from the output report, the multistart solver locates several local optima at each branch. The internal branch-and-bound solver always uses the best known solution at each node. This leads to improved quality of the final integer solution. In order to see the effects of different multistart levels, set the value of *LS_IPARAM_NLP_MAXLOCALSEARCH* macro to lower or higher values and solve the model again.

Example 4: Global Solver with MPI Input Format

This example illustrates the use of LINDO API's global solver to find a global optima to a non-convex model. The model is represented in MPI file format as given below. For details of the MPI file format, see the Instruction-List style interface introduced earlier in this chapter or Appendix D, *MPI File Format*.

```
*  This is a variant of an expression found in
*  Gupta, O. K. and A. Ravindran (1985)
*  "Branch-and-bound Experiments in Convex Nonlinear
*  Integer Programming.", Management Science, 31 pp.1533-1546.
************************************************************
* MODEL:
*
* MIN = x0;
*
* - X1^2*X2 >= -675;
*
* - 0.1*X1^2*X3^2 >= -0.419;
*
*   0.201*X1^4*X2*X3^2 + 100*X0 = 0;
*
* @Bnd(0,X1,1e1);
* @Bnd(0,X2,1e1);
* @Bnd(0,x3,1e1);
```

```
* @Free(x0);
*
* End
************************************************************
BEGINMODEL   gupta21
!  NUMOBJS        1
!  NUMCONS        3
!  NUMVARS        4
VARIABLES
!       Name     Lower Bound   Initial Point     Upper Bound    Type
    X0000000         -1e+030         1.23457          1e+030       C
    X0000001               0         1.23457          1e+001       C
    X0000002               0         1.23457          1e+001       C
    X0000003               0             0.2          1e+001       C
OBJECTIVES
  OBJ00000    MINIMIZE
    EP_PUSH_VAR      X0000000
CONSTRAINTS
  R0000000    G
    EP_PUSH_VAR      X0000001
    EP_PUSH_NUM             2
    EP_POWER
    EP_NEGATE
    EP_PUSH_VAR      X0000002
    EP_MULTIPLY
    EP_PUSH_NUM           675
    EP_NEGATE
    EP_MINUS
  R0000001    G
    EP_PUSH_NUM           0.1
    EP_NEGATE
    EP_PUSH_VAR      X0000001
    EP_PUSH_NUM             2
    EP_POWER
    EP_MULTIPLY
    EP_PUSH_VAR      X0000003
    EP_PUSH_NUM             2
    EP_POWER
    EP_MULTIPLY
    EP_PUSH_NUM         0.419
    EP_NEGATE
    EP_MINUS
  R0000002    E
    EP_PUSH_NUM      0.201
    EP_PUSH_VAR      X0000001
    EP_PUSH_NUM       4
    EP_POWER
    EP_MULTIPLY
    EP_PUSH_VAR      X0000002
    EP_MULTIPLY
    EP_PUSH_VAR      X0000003
    EP_PUSH_NUM       2
    EP_POWER
    EP_MULTIPLY
    EP_PUSH_NUM       100
    EP_PUSH_VAR      X0000000
```

```
    EP_MULTIPLY
    EP_PLUS
    EP_PUSH_NUM       0
    EP_MINUS
ENDMODEL
```

The following C program reads the MPI formatted file above and solves it using LINDO API's global solver.

```
/*
############################################################
#                       LINDO-API
#                    Sample Programs
#                  Copyright (c) 2007 by LINDO Systems, Inc
#
#         LINDO Systems, Inc.           312.988.7422
#         1415 North Dayton St.         info@lindo.com
#         Chicago, IL 60622             http://www.lindo.com
############################################################
  File   : ex_nlp4.c
  Purpose: Read a non-convex nonlinear model from an MPI file and
           optimize with the GOP solver
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
/* LINDO API header file */
#include "lindo.h"
/* Define a macro to declare variables for
    error checking */
#define APIERRORSETUP
   int nErrorCode;
   char cErrorMessage[LS_MAX_ERROR_MESSAGE_LENGTH]
/* Define a macro to do our error checking */
#define APIERRORCHECK
   if (nErrorCode)
   {
      if ( pEnv)
      {
         LSgetErrorMessage( pEnv, nErrorCode,
          cErrorMessage);
         printf("Errorcode=%d:  %s\n", nErrorCode,
          cErrorMessage);
      } else {
         printf( "Fatal Error\n");
      }
      exit(1);
   }
/* main entry point */
/*********************************************************
   Standard callback function to display local and intermediate
   solutions
 *********************************************************/
int  CALLTYPE print_log(pLSmodel model,int iLoc, void *cbData)
{
  static int siter=0,niter=0;
```

```
  static double pobj=0.0;
  static double bestbnd;
  static int status;
  if (iLoc == LSLOC_GOP)
  {
    LSgetCallbackInfo(model,iLoc,LS_IINFO_GOP_STATUS,&status);
    LSgetCallbackInfo(model,iLoc,LS_IINFO_GOP_SIM_ITER,&siter);
    LSgetCallbackInfo(model,iLoc,LS_IINFO_GOP_SIM_ITER,&niter);
    LSgetCallbackInfo(model,iLoc,LS_DINFO_GOP_OBJ,&pobj);
    printf("Iters=%6d \tObj=%11.5e \tStatus=%d\n",siter+niter,pobj,
      status);
  }
  return 0;
} /*print_log*/

int main(int argc, char **argv)
{
   APIERRORSETUP;
   int m, n; /* number of constraints and vars */
   double dObj;
   int    status;
/* declare an instance of the LINDO environment object */
   pLSenv pEnv;
/* declare an instance of the LINDO model object */
   pLSmodel pModel;

  char MY_LICENSE_KEY[1024];

 /*********************************************************
  * Step 1: Create a model in the environment.
  *********************************************************/
  nErrorCode = LSloadLicenseString(
              "../../../license/lndapi60.lic",MY_LICENSE_KEY);

  pEnv = LScreateEnv ( &nErrorCode, MY_LICENSE_KEY);
   if ( nErrorCode == LSERR_NO_VALID_LICENSE) {
      printf( "Invalid License Key!\n");
      exit( 1);
   }
   APIERRORCHECK;
  /*********************************************************
   * Step 2: Create a model in the environment.
   *********************************************************/
   pModel = LScreateModel ( pEnv, &nErrorCode);
   APIERRORCHECK;
  /*********************************************************
   * Step 3: Read the model from an MPS file and get the model size
   *********************************************************/
   nErrorCode = LSreadMPIFile(pModel,"../../mps/testgop.mpi");

   if (nErrorCode != LSERR_NO_ERROR) {
     printf("\n Bad  MPI  format\n");
   } else {
     printf("Reading MPI format. \n\n");
   }
   APIERRORCHECK;
```

```
nErrorCode = LSgetInfo(pModel,LS_IINFO_NUM_VARS,&n);
APIERRORCHECK;
nErrorCode = LSgetInfo(pModel,LS_IINFO_NUM_CONS,&m);
APIERRORCHECK;
/*********************************************************
 * Step 4: Optimize the model
 *********************************************************/
status = LS_STATUS_UNKNOWN;
LSsetModelDouParameter(pModel,LS_DPARAM_CALLBACKFREQ,2.0);
/* Install a callback function */
LSsetCallback(pModel,(cbFunc_t) print_log, NULL);
/* optimize */
printf("\tSolving for Global Solution\n\n");
nErrorCode = LSsolveGOP( pModel, &status);
/*********************************************************
 * Step 5: Access the final solution if optimal or feasible
 *********************************************************/
if (status == LS_STATUS_OPTIMAL ||
    status == LS_STATUS_LOCAL_OPTIMAL ||
    status == LS_STATUS_FEASIBLE )
{
  double *primal = NULL, *dual = NULL;
  int    j, nCont;
  primal = (double *) malloc(n*sizeof(double));
  dual   = (double *) malloc(m*sizeof(double));
  nErrorCode = LSgetInfo(pModel, LS_IINFO_NUM_CONT, &nCont);
  APIERRORCHECK;
  if (nCont < n)
  {
    printf ("\n *** Integer Solution Report *** \n");
    nErrorCode = LSgetInfo(pModel, LS_DINFO_MIP_OBJ, &dObj);
      APIERRORCHECK;
    nErrorCode = LSgetMIPPrimalSolution( pModel,primal);
      APIERRORCHECK;
    nErrorCode = LSgetMIPDualSolution( pModel,dual);
      APIERRORCHECK;
  }
  else
  {
    printf ("\n ***  Solution Report *** \n");
    nErrorCode = LSgetInfo(pModel, LS_DINFO_POBJ, &dObj);
      APIERRORCHECK;
    nErrorCode = LSgetPrimalSolution( pModel,primal);
      APIERRORCHECK;
    nErrorCode = LSgetDualSolution( pModel,dual);
      APIERRORCHECK;
  }
  printf ("\n Objective = %f \n", dObj);
  printf ("\n Primal Solution\n");
  for (j = 0; j<n; j++)
    printf("\tprimal[%d] = %18.10e\n",j, primal[j]);
  printf ("\n Dual Solution\n");
  for (j = 0; j<m; j++)
      printf("\tdual[%d] = %18.10e\n",j, dual[j]);
  free(primal);
  free(dual);
```

```
    }
    /*********************************************************
     * Step 6: Terminate
     *********************************************************/
    nErrorCode = LSdeleteModel( &pModel);
    nErrorCode = LSdeleteEnv( &pEnv);

   /* Wait until user presses the Enter key */
    printf("Press <Enter> ...");
    getchar();
}
```

The source code file for this example may be found in the \lindoapi\samples\c\ex_nlp4\ folder. After creating the executable "ex_nlp4.exe", the application can be run from the DOS-prompt and produce the following report on your screen:

```
C:\lindoapi\samples\c\ex_nlp4>ex_nlp4

Reading MPI format.

        Solving for Global Solution

Iters=     0    Obj=0.00000e+000        Status=12
Iters=     0    Obj=0.00000e+000        Status=3
Iters=     0    Obj=0.00000e+000        Status=12
Iters=     0    Obj=0.00000e+000        Status=12
Iters=     0    Obj=0.00000e+000        Status=12
Iters=     0    Obj=-5.68478e+000       Status=8
Iters=  7330    Obj=-5.68478e+000       Status=8
Iters= 10702    Obj=-5.68478e+000       Status=8
Iters= 13992    Obj=-5.68478e+000       Status=8
Iters= 17454    Obj=-5.68478e+000       Status=8
Iters= 21364    Obj=-5.68478e+000       Status=8
Iters= 24940    Obj=-5.68478e+000       Status=8
Iters= 27064    Obj=-5.68478e+000       Status=8
Iters= 29150    Obj=-5.68484e+000       Status=8
Iters= 36352    Obj=-5.68484e+000       Status=8
Iters= 43502    Obj=-5.68484e+000       Status=8
Iters= 44360    Obj=-5.68484e+000       Status=1

 ***  Solution Report ***

 Objective = -5.684836

 Primal Solution
        primal[0] = -5.6848364236e+000
        primal[1] =  9.9939669649e+000
        primal[2] =  6.7581618276e+000
        primal[3] =  2.0481857461e-001

 Dual Solution
        dual[0] =  8.4219092109e-003
        dual[1] =  1.3567519782e+001
        dual[2] =  1.0000000000e-002
Press <Enter> ...
```

Example 5: Grey-Box Style Interface

This example illustrates the use of LINDO API's grey-box interface. The application reads a nonlinear model in MPI format (i.e. instruction list). Two user-defined functions are provided to enable the EP_USER operators completing the grey-boxes. For details of the MPI file format, see the Instruction-List style interface introduced earlier in this chapter or Appendix D, *MPI File Format*.

```
/*
########################################################################
#                       LINDO-API
#                    Sample Programs
#                  Copyright (c) 2006
#
#         LINDO Systems, Inc.           312.988.7422
#         1415 North Dayton St.         info@lindo.com
#         Chicago, IL 60622             http://www.lindo.com
########################################################################

  @file   : ex_user.c

  @purpose: Solve an NLP that uses two black-box functions within
  the instruction-list interface.

            minimize F(x) = f(x) * x
                G(x) <= 100
             0 <= x  <= 10

  The black-box functions are

    f(x)   the expression sin(pi*x)+cos(pi*x)
    G(x)   the integral[g(x),a,b)], where a,b constants specifying
           the limits of the integral.

  @remark : This application uses the Instruction Style Interface,
  where the instructions are imported from ex_user.mpi file.

  @remark : EP_USER operator is used in the instruction list to
  identify each black-box function and specify the number of
  arguments they take. For each function, the first argument
  is reserved to identify the function, whereas the rest are the
  actual arguments for the associated function.

  @remark : LSsetUsercalc() is used to set the user-defined
  MyUserFunc() function  as the gateway to the black-box functions.

*/


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/* LINDO API header file */
#include "lindo.h"

/* Define a macro to declare variables for
```

```
    error checking */
#define APIERRORSETUP
   int nErrorCode;
   char cErrorMessage[LS_MAX_ERROR_MESSAGE_LENGTH]

/* Define a macro to do our error checking */
#define APIERRORCHECK
   if (nErrorCode)
   {
      if ( pEnv)
      {
         LSgetErrorMessage( pEnv, nErrorCode,
          cErrorMessage);
         printf("nErrorCode=%d:  %s\n", nErrorCode,
          cErrorMessage);
      } else {
         printf( "Fatal Error\n");
      }
      exit(1);
   }

/*************************************************************
 *          Set up an output log function.
 */
static void CALLTYPE print_line(pLSmodel model,
    char *line, void *notting)
{
    if (line)
    {
        printf("%s",line);
    } /*if*/
} /*print_line*/


/*************************************************************
 *         Function g(t) to integrate over [a,b] */
double gox(double x, double t)
{
    double function;
    function = exp(x*cos(t));
    return(function);
}

/*************************************************************
 *          Black-box #2 -- G(x)
 *          Calculated by Simpson's Rule.
 */
double Gox(int n          /* Maximum number of steps (even) n */,
           double x)
{
    int c,k=1;            /* Counters in the algorithm        */
    double a=0;           /* Lower limit x=0                  */
    double b=8*atan(1);   /* Upper limit x=2*pi               */
    double h,dsum;

    dsum=gox(x,a);        /* Initial function value */
```

```
    c=2;
    h=(b-a)/n;            /* Step size h=(b-a)/n */
    while (k <= n-1)      /* Steps through the iteration */
    {
        c=6-c;            /* gives the 4,2,4,2,... */
        dsum = dsum +
          c*gox(x,a+k*h); /* Adds on the next area */
        k++;              /* Increases k value by +1 */
    }
    return ((dsum + gox(x,b))*h/3);
}

/*************************************************************
 *          Black-box function #1 -- f(x).
 */
double fox(double a, double b)
{
  return sin(a) + cos(b);
}

/*************************************************************
 *          Grey-box interface
 */
int CALLTYPE MyUserFunc( pLSmodel model,
    int      nargs,
    double   *argval,
    void     *UserData,
    double   *FuncVal)
{
    double f;
    if (argval[0]==1.) /* argval[0] is the function ID. */
    {
      double a = argval[1];
      double b = argval[2];
      f = fox(a,b);
    }
    else if (argval[0]==2.)
    {
      f = Gox((int)argval[1],argval[2]);
    }

    *FuncVal = f;

    return (0);
} /*print_line*/

/*************************************************************
 *                    Main entry point
 */
int main()

{
    APIERRORSETUP;
    pLSenv pEnv = NULL;
    pLSmodel pModel;
    char MY_LICENSE_KEY[1024];
```

```
/*
* >>> Step 1 <<< Create a LINDO environment.
*/

nErrorCode = LSloadLicenseString(
  "../../../license/lndapi60.lic",MY_LICENSE_KEY);
APIERRORCHECK;

pEnv = LScreateEnv ( &nErrorCode, MY_LICENSE_KEY);
if ( nErrorCode == LSERR_NO_VALID_LICENSE)
{
    printf( "Invalid License Key!\n");
    exit( 1);
}
APIERRORCHECK;

/*
* >>> Step 2 <<< Create a model in the environment.
*/
pModel = LScreateModel(pEnv,&nErrorCode);
APIERRORCHECK;


/*
* >>>> Step 3 <<< Set up the instruction list of the model.
*/
{
    int nLinearz, nAutoDeriv, nConvexRelax, nCRAlgReform;

    /*  Set a log function to call.  */
    nErrorCode =
             LSsetLogfunc(pModel,(printLOG_t) print_line,NULL);
    APIERRORCHECK;

    /* Set linearization level, before a call to LSloadNLPCode.
    * If not specified, the solver will decide */
    nLinearz = 1;
    nErrorCode = LSsetModelIntParameter (pModel,
        LS_IPARAM_NLP_LINEARZ, nLinearz);
    APIERRORCHECK;

    /* Select algebraic reformulation level, convex relaxation*/
    nCRAlgReform = 1;
    nErrorCode = LSsetModelIntParameter (pModel,
        LS_IPARAM_NLP_CR_ALG_REFORM, nCRAlgReform);
    APIERRORCHECK;

    /* Select convex relax level */
    nConvexRelax = 0;
    nErrorCode = LSsetModelIntParameter (pModel,
        LS_IPARAM_NLP_CONVEXRELAX, nConvexRelax);
    APIERRORCHECK;

    /*
    * Set up automatic differentiation before call LSreadMPIFile.
```

```
    * If not specified, the numerical derivative will be applied
    */
    nAutoDeriv = 0;
    nErrorCode = LSsetModelIntParameter (pModel,
        LS_IPARAM_NLP_AUTODERIV, nAutoDeriv);
    APIERRORCHECK;

    /* Set up MyUserFunc() as the user functionas */
    nErrorCode = LSsetUsercalc (pModel,
        (user_callback_t) MyUserFunc, NULL);
    APIERRORCHECK;

    /* Read instructions from an MPI-file */
    nErrorCode = LSreadMPIFile (pModel,"ex_user.mpi");
    APIERRORCHECK;
}


/*
* >>> Step 5 <<< Perform the optimization using the
*                multi-start solver
*/

/* set multi-start as the current NLP solver */
nErrorCode = LSsetModelIntParameter (pModel,
    LS_IPARAM_NLP_SOLVER, LS_NMETHOD_MSW_GRG);
APIERRORCHECK;

nErrorCode = LSoptimize(pModel, LS_METHOD_FREE, NULL);
APIERRORCHECK;


/*
* >>> Step 6 <<< Retrieve the solution
*/
{
    int nLinearity, i, stat, nvars, ncons;
    double objval=0.0, primal[1000];

    /* Get the linearity of the solved model */
    nErrorCode = LSgetModelIntParameter (pModel,
        LS_IPARAM_NLP_LINEARITY, &nLinearity);
    APIERRORCHECK;

    nErrorCode = LSgetInfo(pModel,LS_IINFO_MODEL_STATUS,&stat);
    APIERRORCHECK;
    printf("\n\n\nSolution status = %d \n",stat);

    /* Report the status of solution */
    nErrorCode = LSgetInfo(pModel, LS_IINFO_NUM_VARS,&nvars);
    APIERRORCHECK;

    nErrorCode = LSgetInfo(pModel, LS_IINFO_NUM_CONS,&ncons);
    APIERRORCHECK;

    if (nLinearity)
```

```
        {
            printf("\nModel has been completely linearized.\n");
        }
        else
        {
            printf("\nModel is nonlinear. (nvars=%d, ncons=%d)\n",
                                                    nvars,ncons);
        }

        nErrorCode = LSgetInfo(pModel,LS_DINFO_POBJ,&objval);
        APIERRORCHECK;

        nErrorCode = LSgetPrimalSolution(pModel,primal);
        APIERRORCHECK;

      if (stat==LS_STATUS_OPTIMAL || stat==LS_STATUS_BASIC_OPTIMAL ||
          stat==LS_STATUS_FEASIBLE || stat==LS_STATUS_LOCAL_OPTIMAL)
        {
            printf("\n\nPrinting the solution ... \n\n");
            printf("F(x) = %20.15f \n",objval);
            printf("G(x) = %20.15f \n",Gox(20,primal[0]));
            for (i=0;i<nvars;i++)
              printf("  x  = %20.15f\n",i,primal[i]);
            printf("\n");
        }
        else if (stat == 3)
            printf("\n\nNo feasible solution. \n\n");

        /* Get the linearity of the solved model */
        nErrorCode = LSgetModelIntParameter (pModel,
            LS_IPARAM_NLP_LINEARITY, &nLinearity);
        APIERRORCHECK;

    }

    /*
    * >>> Step 7 <<< Delete the LINDO environment
    */
    LSdeleteEnv(&pEnv);
}
```

The source code file for this example is in the \lindoapi\samples\c\ex_user\ folder. After creating the executable "ex_user.exe", the application can be run from the DOS-prompt and produce the following report on your screen:

```
C:\lindoapi\samples\c\ex_user>ex_user


  Iter   Phase    nInf         Objective         Pinf(sum)       Dinf(rgmax)
     0       0       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     1       0       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     2       0       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     3       3       0  -4.61803483e+000   0.00000000e+000   5.80697188e-001
     4       3       0  -4.61804849e+000   0.00000000e+000   7.11677064e-002
     5       4       0  -4.61804850e+000   0.00000000e+000   2.68772059e-005
     6       4       0  -4.61804850e+000   0.00000000e+000   7.58019439e-009

  Iter   Phase    nInf         Objective         Pinf(sum)       Dinf(rgmax)
     0       0       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     1       0       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     2       0       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     3       3       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     4       3       0   0.00000000e+000   0.00000000e+000   0.00000000e+000

  Iter   Phase    nInf         Objective         Pinf(sum)       Dinf(rgmax)
     0       0       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     1       0       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     2       0       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     3       3       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     4       3       0   0.00000000e+000   0.00000000e+000   0.00000000e+000

  Iter   Phase    nInf         Objective         Pinf(sum)       Dinf(rgmax)
     0       0       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     1       0       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     2       0       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     3       3       0   0.00000000e+000   0.00000000e+000   0.00000000e+000
     4       3       0   0.00000000e+000   0.00000000e+000   0.00000000e+000


Solution status = 8

Model is nonlinear. (nvars=1, ncons=1)


Printing the solution ...

F(x) =    -4.618048495010020
G(x) =    38.603313116588765
  x  =    -0.000000000000006
```

Chapter 8: Stochastic Programming

So far, we worked with deterministic mathematical programs where model parameters (e.g. coefficients, bounds, etc.) are known constants. A stochastic program (SP) is a mathematical program (linear, nonlinear or mixed-integer) in which some of the model parameters are not known with certainty and the uncertainty can be expressed with known probability distributions. Applications arise in a variety of industries:

- Financial portfolio planning over multiple periods for insurance and other financial companies, in face of uncertain prices, interest rates, and exchange rates
- Exploration planning for petroleum companies,
- Fuel purchasing when facing uncertain future fuel demand,
- Fleet assignment: vehicle type to route assignment in face of uncertain route demand,
- Electricity generator unit commitment in face of uncertain demand,
- Hydro management and flood control in face of uncertain rainfall,
- Optimal time to exercise for options in face of uncertain prices,
- Capacity and Production planning in face of uncertain future demands and prices,
- Foundry metal blending in face of uncertain input scrap qualities,
- Product planning in face of future technology uncertainty,
- Revenue management in the hospitality and transport industries.

Multistage Decision Making Under Uncertainty

Stochastic programs fall into two major categories a) Multistage Stochastic Programs with Recourse, and b) Chance-Constrained Programs. With version 6.0, LINDO API's capabilities are extended to solve models in the first category, namely multistage stochastic recourse models. Chance-constrained models will be supported in future versions.

In this chapter, the term 'stochastic program' refers to a multistage stochastic model with recourse. The term 'stage' is an important concept in this chapter. Usually it means the same as 'time period', however there are situations where a stage may consist of several time periods. The terms 'random', 'uncertain' and 'stochastic' are used interchangeably.

Multistage decision making under uncertainty involves making optimal decisions for a T-stage horizon before uncertain events (random parameters) are revealed while trying to protect against unfavorable outcomes that could be observed in the future.

Note: A stage boundary is either a) the beginning of the problem, b) the end of the problem, or c) a point just after a decision event but just before a random event. A stage is the sequence of random events followed by decision events between two successive stage boundaries. Thus, decisions made in stage k are based on all information revealed up to stage k, but no more.

In its most general form, a multistage decision process with $T+1$ stages follows an alternating sequence of random events and decisions. Slightly more explicitly:

0.1) in stage-0, we make a decision x_0, taking into account that…

1.0) at the beginning of stage 1, "Nature" takes a set of random decisions ω_1, leading to realizations of all random events in stage 1, and…

1.1) at the end of stage 1, having seen nature's decision, as well as our previous decision, we make a recourse decision $x_1(\omega_1)$, taking into account that …

2.0) at the beginning of stage 2, "Nature" takes a set of random decisions ω_2, leading to realizations of all random events in stage-2, and…

2.1) at the end of stage 2, having seen nature's decision, as well as our previous decisions, we make a recourse decision $x_2(\omega_1, \omega_2)$, taking into account that …

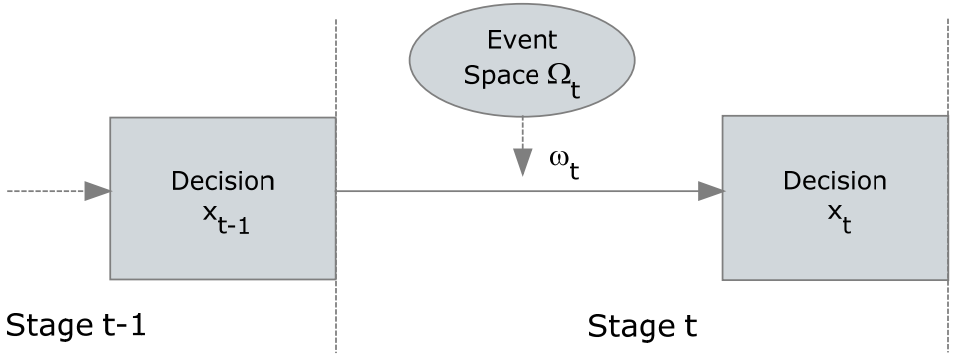
:

:

T.0) At the beginning of stage T, "Nature" takes a random decision, ω_T, leading to realizations of all random events in stage T, and…

T.1) at the end of stage T, having seen all of nature's T previous decisions, as well as all our previous decisions, we make the final recourse decision $x_T(\omega_1,\ldots,\omega_T)$.

This relationship between the decision variables and realizations of random data can be illustrated as follows.



Each decision, represented with a rectangle, corresponds to an uninterrupted sequence of decisions until the next random event. And each random observation corresponds to an uninterrupted sequence of random events until the next decision point.

Recourse Models

The decision taken in stage 0 is called the initial decision, whereas decisions taken in succeeding stages are called 'recourse decisions'. Recourse decisions are interpreted as corrective actions that are based on the actual values the random parameters realized so far, as well as the past decisions taken thus far. Recourse decisions provide latitude for obtaining improved overall solutions by realigning the initial decision with possible realizations of uncertainties in the best possible way.

Restricting ourselves to linear multistage stochastic programs for illustration, we have the following form for a multistage stochastic program with (T+1) stages.

Minimize (or maximize) $c_0x_0 + E_1[c_1x_1 + E_2[\ c_2x_2 \ldots + E_T[\ c_Tx_T\ ] \ldots]]$

Such that

$$
\begin{array}{llll}
A_{00}x_0 & & \sim & b_0 \\
A(\omega_1)_{10}x_0 & + A(\omega_1)_{11}x_1 & \sim & b(\omega_1)_1 \\
A(\omega_1,\ldots,\omega_2)_{20}x_0 & + A(\omega_1,\ldots,\omega_2)_{21}x_1 + A(\omega_1,\ldots,\omega_2)_{22}x_2 & \sim & b(\omega_1,\ldots,\omega_2)_2 \\
 & \vdots \quad \ldots \quad \vdots & & \vdots \\
A(\omega_1,\ldots,\omega_T)_{T0}x_0 & + A(\omega_1,\ldots,\omega_T)_{T1}x_1 + \ldots + A(\omega_1,\ldots,\omega_T)_{TT}x_T & \sim & b(\omega_1,\ldots,\omega_T)_T
\end{array}
$$

$$
\begin{array}{lcl}
L_0 & \leq x_0 \leq & U_0 \\
L(\omega_1)_1 & \leq x_1 \leq & U(\omega_1)_1 \\
\vdots & \vdots & \vdots \\
L(\omega_1,\ldots,\omega_T)_T & \leq x_T \leq & U(\omega_1,\ldots,\omega_T)_T
\end{array}
$$

where, (ω_1, ω_2,…, ω_t) represents random outcomes from event space (Ω_1,..., Ω_t) up to stage t,

$A(\omega_1,\ldots,\omega_t)_{tp}$ is the coefficient matrix generated by outcomes up to stage-t for all p=1…t, t=1…T,

$c(\omega_1,\ldots,\omega_t)_t$ is the objective coefficients generated by outcomes up to stage-t for all t=1…T,

$b(\omega_1,\ldots,\omega_t)_t$ is the right-hand-side values generated by outcomes up to stage-t for all t=1…T,

$L(\omega_1,\ldots,\omega_t)_t$ and $U(\omega_1,\ldots,\omega_t)_t$ are the lower and upper bounds generated by outcomes up to stage-t for all t=1…T,

'$\sim$' is one of the relational operators '$\leq$', '=', or '$\geq$'; and

x_0 and $x_t \equiv x(\omega_1, \omega_2,\ldots, \omega_t)_t$ are the decision variables (unknowns) for which optimal values are sought. The expression being optimized is called the cost due to initial-stage plus the expected cost of recourse.

Note:	LINDO API can solve linear, nonlinear and integer multistage stochastic programming problems.

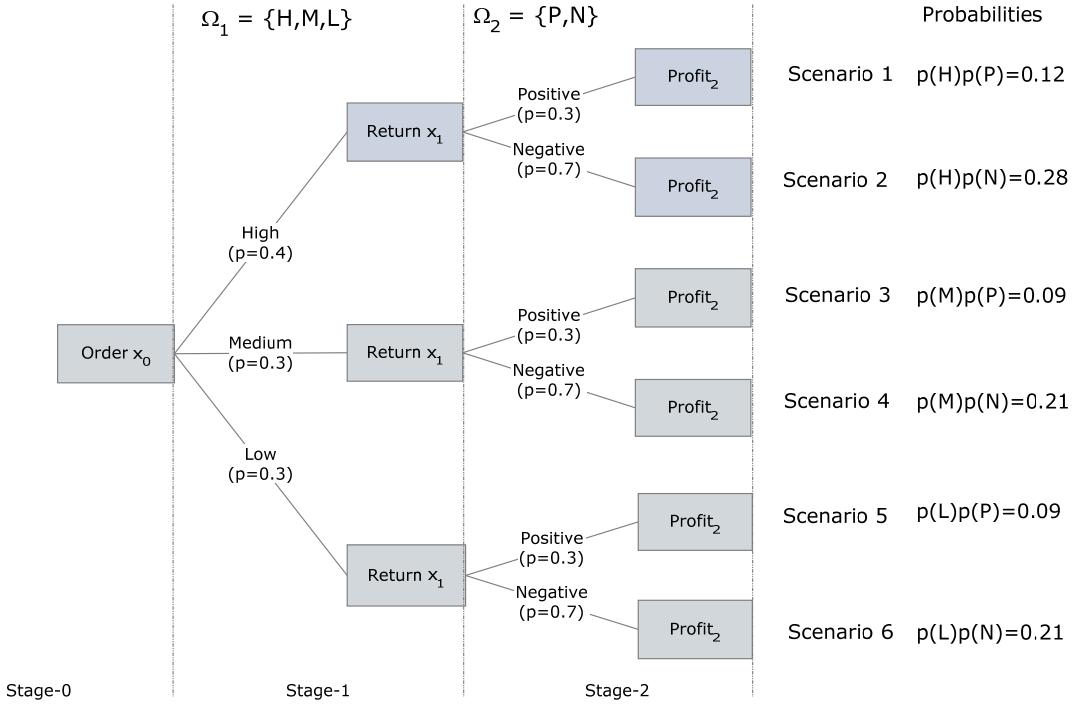
Scenario Tree

When the probability distributions for the random parameters (events) are discrete, there are only a finite number of outcomes in each stage. With each random parameter fixed to one of its possible

outcomes, one can create a scenario representing one possible realization of the future. Enumeration of all possible combinations of outcomes allows us to represent all scenarios in a tree, with each scenario being a path from the root of the tree to one of its leaves. The nodes visited by each path correspond to values assumed by random parameters in the model.

We illustrate the construction of a scenario tree with a stochastic version of the well-known Newsvendor inventory problem. In this problem, we must decide how much to order initially and then later, how much of any unsold product to return before the end of the planning horizon. There is a shortage penalty when there are lost sales and a carrying cost for left over units. The decision process takes place under uncertain demand and uncertain price per returned item:

- In stage 0, the order quantity has to be decided (under uncertain demand).
- In stage 1, at the beginning, the demand is revealed. A recourse decision, at the end of stage 1, is the number of units to be returned to the publisher (for an uncertain refund price)
- In stage 2 at the beginning, the refund price is announced by the publisher. The price per returned item can be either
    - Positive (i.e. publisher accepts them at a high price which covers the cost of shipping and handling) or
    - Negative (i.e. publisher accepts them at a low price which doesn't cover the cost of shipping and handling).
- The objective is to maximize the total expected profit at the end of planning horizon (stage 2).



In the scenario tree above, x_0 represents the initial decision, order size to be determined before seeing any of the random outcomes. x_1 represents the quantity to return to the publisher of any portion of the unsold units. *Profit*$_2$ represents the total profit collected at the end of planning horizon. The notation Ω_1 represents the event space for the unknown demand, for which there are three different possible outcomes Ω_1 = {Low, Medium, and High} with probabilities {0.4, 0.3, 0.3}, respectively. Once we observe the demand $\omega_1 \in \Omega_1$, we make a recourse decision x_1 based upon which ω_1 nature chose and our previous decision x_0. The notation Ω_2 represents the event space for refund price per unsold newspapers if returned to the publisher in stage 2. This event has two different outcomes Ω_2 = {Positive, Negative} with probabilities {0.7, 0.3}. Once the refund price $\omega_2 \in \Omega_2$ in stage 2 is observed, the total profit would be computed by the model as the final decision *Profit*$_2$.

It should be clear from the scenario tree that,

- There are as many distinct scenarios in the SP as there are leaf-nodes.
- Each root-leaf path defines a scenario, induced by a full observation of all random events.
- There is a one-to-one correspondence between the scenarios and the leaf-nodes in the tree.
- The unconditional probability of a node is computed by multiplying the conditional probabilities of the nodes positioned on the path, which starts from the root and terminates at that node.
- The unconditional probability of each leaf-node corresponds to the probability of the associated scenario.
- Each node in the tree corresponds to a vector of random parameter with a particular history up to that node in some scenario.
- The branches out of each node enumerate all possible outcomes associated with random parameters associated with it in order to construct the history of random parameters that belong to next stage.

Setting up SP Models:

Setting up an SP model in the LINDO API involves three major steps in the given order:

a) Specify the core model as if all of nature's decisions are known. This is simply describing the mathematical relations among all the variables in a typical mathematical model as described in Chapters 3 through 7. If an instruction list is used to represent the core model, the EP_PUSH_SPAR instruction is required in place of those LS_PUSH_NUM to identify the parameters that are in fact stochastic. If the core model will be set up using the standard array representation, a dummy nonzero element is required for each random parameter as a placeholder.
b) Provide the time structure. This involves listing, either explicitly or implicitly, the stage of every random parameter, variable and constraint in the model.
c) Provide the distributions describing the random parameters and the type of sampling from the underlying distributions, when or if required.

An alternative way of inputting an SP to the LINDO API is via files. To achieve this, one must prepare at least three files each of which will assume the role of the tasks above:

1. A "core" or deterministic version of the model in one of the file formats supported by LINDO API, such as an MPI file, LINDO file or MPS file. If MPS or LINDO file formats are used, the core model must be in temporal order and each random parameter must have a dummy

(possibly an average) value in the core model to serve as a placeholder. The file extension is either .mpi or .mps (or .ltx) depending on the format preferred.

2. A stage or time file with a suffix of .time, which associates each constraint and variable with a stage,
3. A stochastic data file with a suffix of .stoch, which provides the information about all random parameters and their properties.

The three-file input is collectively called the SMPI or SMPS file format. The details on the format are summarized in Appendices E and F. The contents of these files correspond almost exactly with the contents of the data objects used to set up the SP programmatically given in the following sections.

Loading Core Model:

Consider the Newsvendor problem written as a deterministic linear program after fixing the random parameters to dummy values temporarily.

```
! Stochastic Newsvendor Model;
DATA:
 C = 30;  ! Purchase cost/unit;
 P = 5;   ! Penalty shortage cost/unit unsatisfied demand;
 H = 10;  ! Holding cost/unit leftover;
 V = 60;  ! Revenue per unit sold;

! Random demand (D);
 D = 63;
! Random refund per return;
 R = 9;
ENDDATA

MAX = Z;
! Units bought, X, Buy at least 0 (serves as a dummy constraint for
stage 1);
[Row1] X >= 1;
! Inventory (I) and Lost Sales (L);
[Row2]  I = X + L - D;
! Units sold S, and inventory left over, I;
[Row3]  S = X - I;
! Y units returned to vendor for a possible refund, and E kept;
[Row4]  Y + E = I;
! Profit, to be maximized;
[Profit]  Z = V*S - C*X - H*I - P*L + Y*R - H*E;
```

Using Instruction Lists

Starting with the deterministic version given above, we rewrite the model in instruction list format and then mark each random parameter (*D* and *R*) by replacing the associated EP_PUSH_NUM instruction with an EP_PUSH_SPAR instruction. This is illustrated in the following where the stochastic parameters *R* and *D* are marked in red:

```
[Row2] I = X + L – D
```

Deterministic			Stochastic		
Row2	E		Row2	E	
	EP_PUSH_VAR	I		EP_PUSH_VAR	I
	EP_PUSH_VAR	X		EP_PUSH_VAR	X
	EP_PUSH_VAR	L		EP_PUSH_VAR	L
	EP_PLUS			EP_PLUS	
	EP_PUSH_NUM	**63**	⟷	**EP_PUSH_SPAR**	**D**
	EP_MINUS			EP_MINUS	
	EP_MINUS			EP_MINUS	

```
[Profit] Z = V*S - C*X - H*I - P*L + Y*R - H*E;
```

Deterministic			Stochastic		
PROFIT	E		PROFIT	E	
	EP_PUSH_VAR	Z		EP_PUSH_VAR	Z
	EP_PUSH_NUM	60		EP_PUSH_NUM	60
	EP_PUSH_VAR	S		EP_PUSH_VAR	S
	EP_MULTIPLY			EP_MULTIPLY	
	EP_PUSH_NUM	30		EP_PUSH_NUM	30
	EP_PUSH_VAR	X		EP_PUSH_VAR	X
	EP_MULTIPLY			EP_MULTIPLY	
	EP_MINUS			EP_MINUS	
	EP_PUSH_NUM	10		EP_PUSH_NUM	10
	EP_PUSH_VAR	I		EP_PUSH_VAR	I
	EP_MULTIPLY			EP_MULTIPLY	
	EP_MINUS			EP_MINUS	
	EP_PUSH_NUM	5		EP_PUSH_NUM	5
	EP_PUSH_VAR	L		EP_PUSH_VAR	L
	EP_MULTIPLY			EP_MULTIPLY	
	EP_MINUS			EP_MINUS	
	EP_PUSH_VAR	Y		EP_PUSH_VAR	Y
	EP_PUSH_NUM	**9**	⟷	**EP_PUSH_SPAR**	**R**
	EP_MULTIPLY			EP_MULTIPLY	
	EP_PLUS			EP_PLUS	
	EP_PUSH_NUM	10		EP_PUSH_NUM	10
	EP_PUSH_VAR	E		EP_PUSH_VAR	E
	EP_MULTIPLY			EP_MULTIPLY	
	EP_MINUS			EP_MINUS	
	EP_MINUS			EP_MINUS	

As discussed in Chapter 7, `EP_PUSH_NUM` instruction loads (pushes) a constant value onto the top of stack, whereas `EP_PUSH_SPAR` instruction loads the name of the random parameter on the top of the stack. An appropriate index for each stochastic parameter will be created. Normally, the index value depends on the order it appears in the instruction list. Finally, the core model is loaded by calling the LSloadInstruct() function in the usual way.

Note: When the core model is loaded with an instruction list using LSloadInstruct(), all stochastic parameters will automatically be assigned a unique index. This index can be used to access to all information about that stochastic parameter, such as its stage index, stochastic data and others. See Chapter 2 for the public functions that relies on this index.

Using the Array Representation of a Model:

Because our model is linear, it could also be described in standard array representation (also called the matrix form). Refer to Chapter 1 for an overview. Starting with the deterministic version in matrix form, we have

	X (0)	I (1)	L (2)	S(3)	Y(4)	E (5)	Z (6)		
Max							1		
Row1 (0)	1							>	1
Row2 (1)	-1	1	-1					=	**-63**
Row3 (2)	-1	1		1				=	0
Row4 (3)		-1			1	1		=	0
Profit(4)	-30	-10	-5	60	**-9**	-10	-1	=	0

The indices of variables and constraints are given next to their names in parenthesis. The equivalent array representation, where stochastic parameters are marked in red, is

```
Column-indices:  0          1          2     3      4     5      6
Column-starts:  [0          4          8    10     12    14     16   17]
Values:         [1 -1 -1 -60 1 -1 -1 10 -1 -5  1  60  1 -9  1 -10 -1]
Row-index:      [0  1  2   4 1  2  3  4  1  4  2   4  3  4  3   4  4]

Right-hand side values = [ 1 -63 0 0 0 ]
Objective coefficients = [ 0 0 0 0 0 0 1 ]
Constraint senses = [ G E E E E]
Lower bounds = [ 0 0 0 0 0 0 0]
Upper bounds = [ ∞ ∞ ∞ ∞ ∞ ∞ ∞]
```

Take note of the coordinates of the matrix and vector elements where stochastic parameters are located. The random parameters in matrix form are identified by the (row, column) indices of the variable they go with. This information uniquely identifies a random element in matrix form and is needed when loading the stochastic data associated with each random parameter. For instance, in this example

- The LP matrix has one stochastic element at matrix index position `(4,4)`.
- The RHS has one stochastic element at index 1.

Note: When the LP data contains a random parameter, the row and column indices of the variable and constraint that goes with the random parameter is sufficient to identify it. If the random parameter is in the OBJECTIVE row, the RHS column or the BOUNDS rows then a dummy index is needed to identify these vectors. The following macros identify such dummy rows and columns:

LS_JCOL_RHS: Stochastic parameter is a RHS value (belongs to RHS column)

LS_IROW_OBJ: Stochastic parameter is an objective coefficient (belongs to OBJ row)

LS_IROW_VUB: Stochastic parameter is a lower bound (belongs to LO row)

LS_IROW_VLB: Stochastic parameter is an upper bound (belongs to UP row)

LS_IROW_VFX: Stochastic parameter is a fixed bound (belongs to FX row)

LS_JCOL_INST: Stochastic parameter is an instruction code.

The important point is that each stochastic element has a nonzero position reserved in the constraint matrix and/or vector, possibly with a default or dummy value. Once this is ensured, the core model could be loaded by calling the LSloadLPData() function in the usual sense.

Note: In order to load the core model in matrix form using LSloadLPData, the constraints and variables in the core model should be in ascending order with respect to their stages. LP data which has this property is said to have temporal ordering. Such a requirement doesn't exist when loading the core model via an instruction list.

Loading the Time Structure:

Loading timing or staging information tells LINDO API a) how many time stages there are and b) the time stage of each random parameter, variable and constraint. It is convenient to give a label to each time stage just like we do for variables and constraints.

For this particular example, there are three stages, labeled `TIME1`, `TIME2`and `TIME3`, and they are associated with random parameters, variables and constraints as summarized in the following table.

Variables	**Index**	**Time Stage**	**Stage Index**
X	0	TIME1	0
I	1	TIME2	1
L	2	TIME2	1
S	3	TIME2	1
Y	4	TIME2	1
E	5	TIME2	1
Z	6	TIME3	2
Constraints			
Row1	0	TIME1	0
Row2	1	TIME2	1
Row3	2	TIME2	1
Row4	3	TIME2	1
Profit	4	TIME3	2
Random Par.			

D	0	TIME2	1
R	1	TIME3	2

Sometimes it may not be easy to deduce the stages of constraints involving several variables from different stages. The general rule is to set the stage index of the constraint to the largest of the variable stage indices which appear in that constraint.

A typical call sequence that loads time/stage structure is as in the following code snippet in C language. See lindoapi/samples/c/ex_sp_newsboy directory for the complete application.

```
{ /* Load stage data */
  int      errorcode   = LSERR_NO_ERROR;
  int      numStages   = 3;
  int      colStages[]   = {0,  1,  1,  1,  1,  1, 2}; /* Stage
indices of columns */
  int      rowStages[]   = {0,  1,  1,  1, 2 };         /* Stage
indices of rows */
  int      panSparStage[]= {1,  2 }; /* Stage indices of stochastic
parameters */

  errorcode=LSsetNumStages(pModel,numStages);
  if (errorcode!=LSERR_NO_ERROR)
{fprintf(stdout,"\nError=%d\n",errorcode); exit(1);}

  errorcode=LSloadVariableStages(pModel,colStages);
  if (errorcode!=LSERR_NO_ERROR)
{fprintf(stdout,"\nError=%d\n",errorcode); exit(1);}

  errorcode=LSloadConstraintStages(pModel,rowStages);
  if (errorcode!=LSERR_NO_ERROR)
{fprintf(stdout,"\nError=%d\n",errorcode); exit(1);}

  errorcode=LSloadStocParData(pModel,panSparStage,NULL);
  if (errorcode !=0) { fprintf(stdout,"\nError=%d\n",errorcode);
exit(1);}
}
```

Temporal Time Structure:

If the core model is represented in matrix form and loaded with `LSloadLPData()`, it is required to have the so-called temporal ordering. When the LP data has temporal ordering, time structure can be represented by specifying the indices (or names) of the first variable and constraint in each stage.

In Newsvendor problem, the model is (already) in temporal order, thus it is sufficient to specify the indices of the first constraint and variable in each stage.

Variables	Index	Time Stage	Stage Index
X	0	TIME1	0
I	1	TIME2	1
Z	2	TIME3	2

Constraints			
Row1	0	TIME1	0
Row2	1	TIME2	1
Profit	2	TIME3	2

As seen in the table, variable *I* is the first variable in stage 2, and due to temporal ordering, all variables up to the next variable (*Z*) are also in stage 2. Similarly, *Row2* is the first constraint in stage 2 and all constraints up to the next row *Profit* belong to stage 2.

Note: Currently, temporal time structures can only be loaded through SMPS formatted files. Public API functions will be made available in future releases.

Loading the Stochastic Structure:

The final step of loading an SP model is to specify the stochastic data associated with all random parameters. This can be achieved in three different ways depending on the type of randomness and their relation with each other. The random parameters can be

1. Independently distributed: when the behavior of the system in some stage depends on each random parameter in that stage independently. Such parameters can be represented in two forms:
    a. A univariate parametric distribution. The distribution can be continuous (e.g. univariate Normal distribution) or discrete (e.g. Poisson distribution). The stochastic data for such parameters can be loaded to LINDO API via `LSaddParamDistIndep()` function.
    b. A univariate discrete distribution in the form of a table describing the range of values that the random parameter can take and the probabilities associated with each. The stochastic data for such parameters can be loaded to LINDO API via `LSaddDiscreteIndep()` function.
2. Jointly distributed: when the behavior of the system in some stage depends on two or more interdependent random parameters in that stage. Such random parameters and the relationships among them can be represented in two forms:
    a. A continuous joint distribution function (e.g. multivariate normal distribution). Multivariate continuous distributions cannot be loaded explicitly. The user is expected to load each parameter as a univariate continuous parameter and then add an appropriate correlation structure via `LSloadCorrelationMatrix()`.
    b. A discrete joint distribution table specifying the probabilities of each joint realization of a vector of random parameters. The representation is similar to the univariate case except that each event ω is a vector and the event space Ω is a set of vectors with known probabilities. The stochastic data for such parameters can be loaded to LINDO API via `LSaddDiscreteBlocks()` function.

3. Distributed with interstage dependency: when the event space Ω in some stage depends on the realizations of random parameters in previous stages. A typical example is when modeling the operations of an investment bank at a particular stage. It may encounter different event spaces in the future depending on the past decisions that led to a particular state. For instance, a set of decisions might lead to bankruptcy beyond which future events will be totally different than those in non-bankruptcy scenarios. The stochastic data for such cases can be loaded to LINDO API via `LSaddScenario()` function

Note: In cases where random parameters don't have interstage dependency, the stochastic data can be loaded for each stage separately and the scenario tree can be created by LINDO API automatically. When there is interstage dependency, the user is expected to create the scenario tree explictly by loading scenarios via LSaddScenario().

Typical usage of these functions is illustrated for the Newsvendor problem under various stochastic data types. It is assumed that an instruction list has been used to load the core model.

Case 1: Let *D* and *R* be independently distributed discrete parameters with the following event space and event probabilities:

Random Param.	Index	Ω	P(ω)	|Ω|
D	0	{H=90,M=60,L=30}	{0.4, 0.3, 0.3}	3
R	1	{P=9,N=-15}	{0.7, 0.3}	2

These data can be loaded to LINDO API as in the following code snippet in C language. See lindoapi/samples/c/ex_sp_newsboy directory for the complete application modeling this case.

```
{ /* Load discrete independent variables */
  int      errorcode = 0;
  int      iRow      = -99;
  int      jCol      = -99;

  // declarations for stochastic parameter D (index=0)
  int      iStv0     = 0;                 // index of stoc. param.
  int      nRealizations0 = 3;            // size of event space
  double   padVals0[]  = {90,      60,     30};    // event space
  double   padProbs0[] = {0.4,    0.3,     0.3};   // probabilities of
                                                    //events

  // declarations for stochastic parameter R (index=1)
  int      iStv1      = 1;               // index of stoc. param.
  int      nRealizations1 = 2;           // size of event space
  double   padVals1[]  = {9,      -15};  // event space
  double   padProbs1[] = {0.3,    0.7};  // probabilities of events

  // load stoc. param. 0
  errorcode=LSaddDiscreteIndep(pModel,iRow,jCol,iStv0,
nRealizations0,padProbs0,padVals0,LS_REPLACE);
  if (errorcode !=0) { fprintf(stdout,"\nError=%d\n",errorcode);
exit(1);}

  // load stoc. param. 1
  errorcode=LSaddDiscreteIndep(pModel,iRow,jCol,iStv1,
nRealizations1,padProbs1,padVals1,LS_REPLACE);
  if (errorcode !=0) { fprintf(stdout,"\nError=%d\n",errorcode);
exit(1);}

}
```

Case 2: Let *D* and *R* be independently normal distributed with distribution parameters (μ=45, σ=10) and (μ= -3, σ=2), respectively.

Random Param.	Index	Distribution	|Ω|
D	0	NORMAL(45,10)	+inf
R	1	NORMAL(-3,2)	+inf

This data can be loaded to LINDO API as in the following code snippet in C language. See lindoapi/samples/c/ex_sp_newsboy directory for the complete application which models this case.

```
{ /* Load discrete independent variables */
  int      errorcode = 0;
  int      iRow      = -99;
  int      jCol      = -99;

  // declarations for stochastic parameter D (index=0)
  int      iStv0     = 0;                    // index of stoc. param.
  int      nDistType0 = LSDIST_TYPE_NORMAL;  // type of distribution
  int      nDistParams0 = 2;           // number of distrib. params.
  double   padParams0[] = {45,     10}; //distrib. params (mu, sigma)

  // declarations for stochastic parameter R (index=1)
  int      iStv1      = 1;                // index of stoc. param.
  int      nDistType1 = LSDIST_TYPE_NORMAL; // type of distribution
  int      nDistParams1 = 2;           // number of distrib. params.
  double   padParams1[] = {-3,    2}; // distrib. params (mu, sigma)

  // load stoc. param. 0
  errorcode=LSaddParamDistIndep(pModel,iRow,jCol,iStv0,
nDistType0,nDistParams0,padParams0,LS_REPLACE);
  if (errorcode !=0) { fprintf(stdout,"\nError=%d\n",errorcode);
exit(1);}

  errorcode=LSaddParamDistIndep(pModel,iRow,jCol,iStv1,
nDistType1,nDistParams1,padParams1,LS_REPLACE);
  if (errorcode !=0) { fprintf(stdout,"\nError=%d\n",errorcode);
exit(1);}

}
```

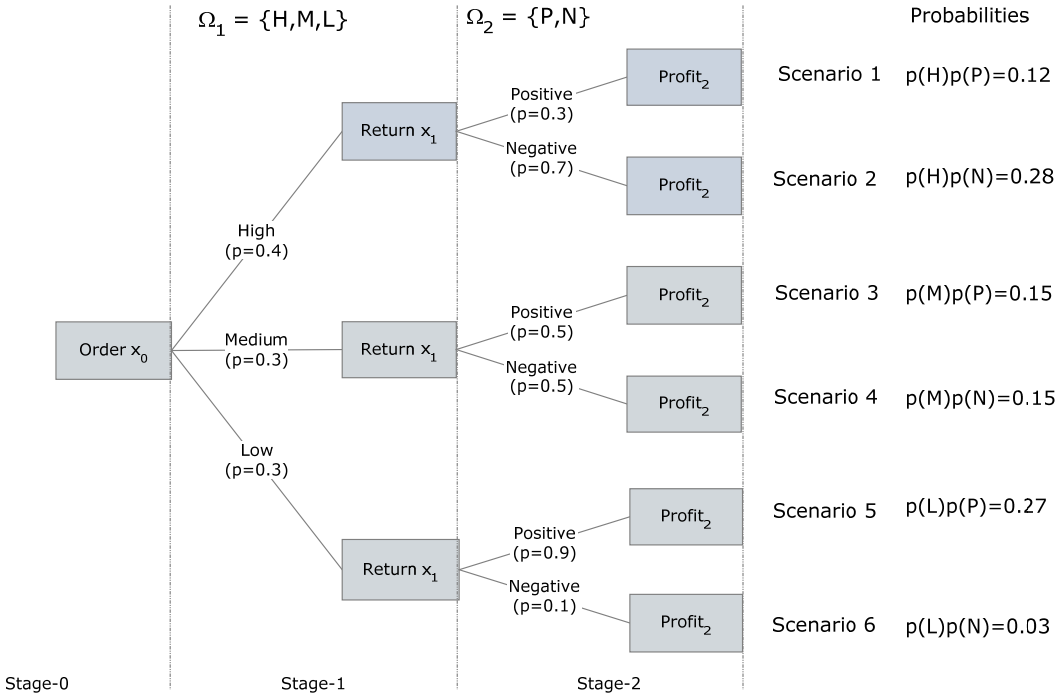
Note: It is possible to have a mixed case, where *D* is discrete and *R* is continuous. In such a case, declarations for *D* from case 1 and declarations for *R* from case 2 will be used along with associated function calls.

Case 3: Consider a case where probabilities of *R* are dependent of the observed value of *D* in the previous stage. This is a situation where *D* and *R* are distributed with interstage dependency. For the Newsvendor problem, suppose we have the following joint distribution table.

Stage-1	**Stage-2**	
D	**R**	**Prob(D,R)**
(H)igh (90)	(P)ositive (9)	0.12
	(N)egative (-15)	0.28
(M)edium (60)	(P)ositive (9)	0.15
	(N)egative (-15)	0.15
	(P)ositive (9)	0.27

(L)ow (30)		
	(N)egative (-15)	0.03

An equivalent scenario tree will look like:



In such a case, the scenarios should be explicitly loaded to LINDO API as in the following code snippet in C language. See lindoapi/samples/c/ex_sp_newsboy directory for the complete application which models this case.

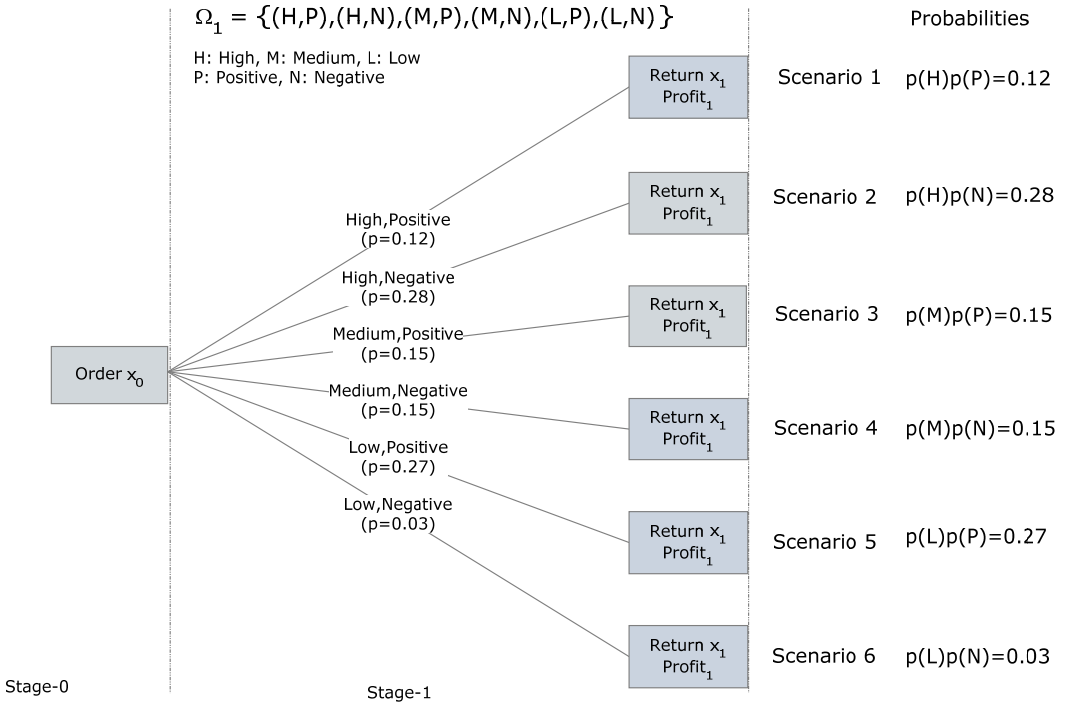
```
/* Load scenario 1 */
{ int      errorcode   = 0;
  int      iStage    = 1   , jScenario = 0     , iParentScen=-1;
  int      nElems    = 2   , paiStvs[] = {0, 1};
  double   dProb     = 0.12, padVals[] = {90, 9};
  errorcode=LSaddScenario(pModel,jScenario,iParentScen,
       iStage,dProb,nElems,NULL,NULL,paiStvs,padVals,LS_REPLACE);
}

/* Load scenario 2 */
{ int      errorcode   = 0;
  int      iStage    = 2   , jScenario = 1     , iParentScen=0;
  int      nElems    = 1   , paiStvs[] = {1};
  double   dProb     = 0.28, padVals[] = {-15};
  errorcode=LSaddScenario(pModel,jScenario,iParentScen,
       iStage,dProb,nElems,NULL,NULL,paiStvs,padVals,LS_REPLACE);
}

:
:
:

/* Load scenario 6 */
{ int      errorcode   = 0;
  int      iStage    = 2   , jScenario = 0     , iParentScen=-1;
  int      nElems    = 1   , paiStvs[] = {1};
  double   dProb     = 0.03, padVals[] = {-15};
  errorcode=LSaddScenario(pModel,jScenario,iParentScen,
       iStage,dProb,nElems,NULL,NULL,paiStvs,padVals,LS_REPLACE);

  if (errorcode !=0) { fprintf(stdout,"\nError=%d\n",errorcode);
exit(1);}
}
```

Case 4: Consider a new case where the Newsvendor model has two stages and the random parameters *D* and *R* belong to the same stage with the following time structure

Variables	Index	Time Stage	Stage Index
X	0	TIME1	0
I	1	TIME2	1
L	2	TIME2	1
S	3	TIME2	1
Y	4	TIME2	1
E	5	TIME2	1
Z	6	TIME2	2
Constraints			
Row1	0	TIME1	0
Row2	1	TIME2	1
Row3	2	TIME2	1
Row4	3	TIME2	1
Profit	4	TIME2	1
Random Par.			

D	0	TIME2	1
R	1	TIME2	1

After the new time structure is loaded to LINDO API, we can work out the loading of the stochastic data as follows. Suppose the joint distribution probabilities are the same as case 3. This leads to the following scenario tree.



Note: This new version of Newsvendor problem is actually a special case of the original problem, where stage 1 and stage 2 (TIME2 and TIME3) are aggregated into a single stage which is now called stage 1 (TIME2). The consequences of this aggregation are that all random parameters, constraints and variables that belonged to stage 1 and stage 2 in the original problem now belong to stage 1 in the aggregated version.

As it can be seen in the scenario tree, each outcome in stage 1 corresponds to a block realization of a vector of random parameters, namely *D* and *R*. The associated stochastic data can be loaded to LINDO API as in the following code snippet in C language. See lindoapi/samples/c/ex_sp_newsboy directory for the complete application modeling this case.

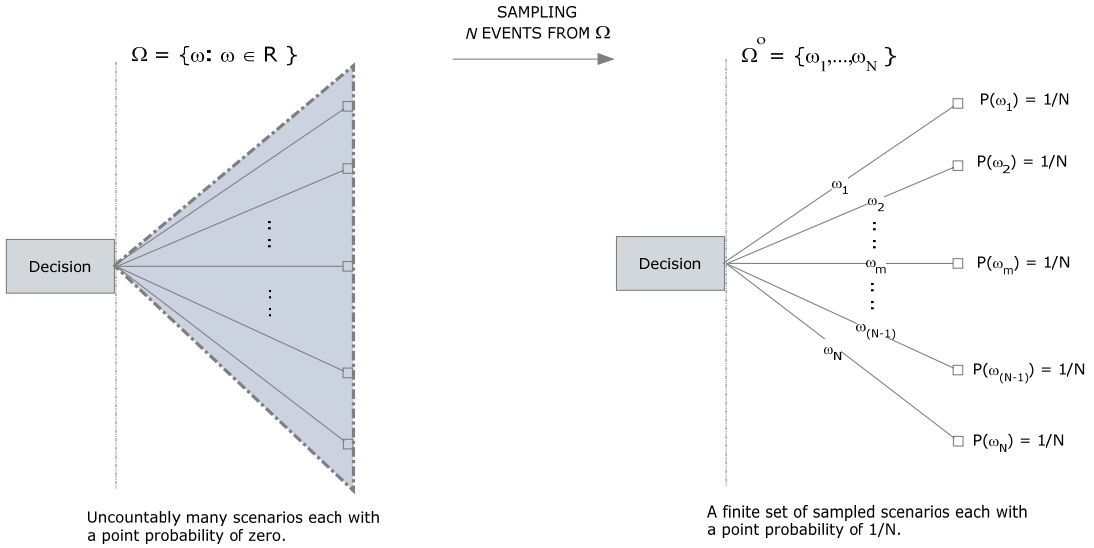
Note: Case 4 is a relaxation of case 3 because of the (implicit) non-anticipativity constraints in case 3. In terms of this particular example, case 4 imposes no extra restrictions on stage 1 variables (quantity returned to the vendor) because the refund price is announced prior to stage 1 decisions are taken.

```
{ /* Load a single block */
  int      errorcode   = 0;
  int      iStage      = 1;
  int      nBlockEvents= 6;
  int      pakStart[] =  { 0,      2,      4,      6,      8,     10,
12};
  int      paiStvs[]  =  { 0, 1,   0, 1,   0, 1,   0, 1,   0, 1,   0,
1};
  double   padVals[]  =  { 90,9,  90,-15,  60,9,  60,-15,  30,9,
30,-15};

  double   padProb[]  =  { 0.12,  0.28,    0.15,  0.15,    0.27,
0.03 };

  errorcode=LSaddDiscreteBlocks(pModel,iStage,nBlockEvents,
        padProb,pakStart,NULL,NULL,paiStvs,padVals,LS_REPLACE);
  if (errorcode !=0) { fprintf(stdout,"\nError=%d\n",errorcode);
exit(1);}
} // end-block
```

Monte Carlo Sampling

In stochastic programming where one or more stochastic parameters have continuous or discrete but infinite event space, there will be too many scenarios making the model computationally intractable. For such cases Monte Carlo sampling (also called pre-sampling) can be used to approximate the problem to work with a finite scenario tree. As illustrated in the figure below, if the model has a single stochastic parameter with a continuous distribution such as the Normal Distribution; one can discretize the event space simply by generating *N* sample points and construct a finite and tractable scenario tree. This is also true for discrete distributions with infinite event space like the Poisson distribution.

Note: Sampling a scenario tree prior to the optimization process is also called pre-sampling. This is to distinguish this type of sampling from the one that is used during optimization process. In LINDO API, sampling refers to pre-sampling unless otherwise is stated.



Note: Since the point probability of each scenario in the original model is zero, it is customary to set the probabilities of sampled scenarios to 1/*N*. However, the user can always define customized sampling approaches to work with different scenario probabilities.

Given the parametric distribution of each stochastic parameter, LINDO API's sampling routines can be used to generate univariate samples from these distributions efficiently. The user has the option to use antithetic-variates or Latin-hyper-square sampling to reduce the sample variance. See Appendix 8c at the end of this chapter for a brief definition of these techniques. Appendix 8b gives a general account of pseudo-random number generation in LINDO API.

After the samples are created, the sample points could be used to define discrete distributions, which will approximate the original distribution. Repeating this for all continuous stochastic parameters, one could reformulate the model as in case 1 or extend it to cases 3 and 4 discussed above.

1. Sampling from a univariate distribution should follow the steps below. Create a sample object by calling `LSsampCreate()` function specifying the parametric distribution type. See Chapter 2 for a list of supported distributions.
2. Set the parameters of the distribution associated with the sample object.
3. Create a pseudorandom generator object by calling `LScreateRG`() function and specify its seed for initialization.
4. Assign the random generator to the sample object by calling `LSsampSetRG`() function.
5. Generate desired number of sample points by calling LSsampGenerate() specifying the variance reduction method to be used.
6. Retrieve the sample points generated by calling LSsampGetPoints().

The following code snippet illustrates this process in C language. See lindoapi/samples/c/ex_dist_gen directory for the complete application.

```
{
   pSample = LSsampCreate(pEnv, LSDIST_TYPE_NORMAL, &nErrorCode);

   // Set two parameters to define the normal distribution
   nErrorCode = LSsampSetDistrParam(pSample, 0,dMean);
   nErrorCode = LSsampSetDistrParam(pSample, 1,dSigma);

   // Create and assign a random number generator (RG)
   pRG = LScreateRG(pEnv, LS_RANDGEN_FREE);
   nErrorCode = LSsampSetRG(pSample,pRG);
   LSsetRGSeed(pRG, 1031);

   // Generate 30 random points with LHS variance reduction in charge
   fprintf(stdout,"\nGenerating %d random variables...\n",30);
   nErrorCode = LSsampGenerate(pSample, LS_LATINSQUARE, 30);
   nErrorCode = LSsampGetPoints(pSample,&i,&pX);

}
```

Generating dependent samples

In certain situations, the modeler may require some of the samples to be dependent to each other. It is common to characterize such dependencies by standard correlation measures, like

- Pearson's linear correlation.
- Spearman's rank correlation.
- Kendall's rank correlation.

For definitions of these correlation types, refer to Appendix 8a at the end of this chapter.

LINDO API allows the users to generate dependent samples by the simple steps below.

1. Create independent univariate sample objects and generate samples of equal size as described above. The sample size should be greater than or equal to the number of sample objects.
2. Define the lower or upper triangular part of the target correlation matrix Q in sparse form. Its size should be equal to the number of sample objects (i.e. the dimension of the multivariate sample).
3. Load the correlation matrix by calling `LSsampInduceCorrelation`() function.
4. Retrieve the correlation induced (CI) sample points by `LSsampGetCIPoints`() function.

The following code snippet illustrates this process in C language. See lindoapi/samples/c/ex_sp_corr directory for its application in SP context.

```
{
   nDim = 3;
   // Create a common random number generator.
   pRG = LScreateRG(pEnv, LS_RANDGEN_FREE);
   LSsetRGSeed(pRG, 1031);

   // Create nDim sample objects and generate 30 sample points for
   // each.
   for (i=0; i< nDim; i++)
   {
       paSample[i] = LSsampCreate(pEnv, LSDIST_TYPE_NORMAL,
       &nErrorCode);

     // Set two parameters to define the normal distribution
     nErrorCode = LSsampSetDistrParam(pSample[i], 0,dMean);
     nErrorCode = LSsampSetDistrParam(pSample[i], 1,dSigma);

     // Assign the common random number generator (RG)
     nErrorCode = LSsampSetRG(pSample[i],pRG);

     // Generate 30 random points with LHS variance reduction in
     // charge
     fprintf(stdout,"\nGenerating %d random variables...\n",30);
     nErrorCode = LSsampGenerate(pSample[i], LS_LATINSQUARE, 30);
   }

   // Induce Pearson correlations to the original sample
   {
     int TargetQCnonzeros = 6;
     int TargetQCvarndx1[] = {0, 0, 0, 1, 1, 2};
     int TargetQCvarndx2[] = {0, 1, 2, 1, 2, 2};
     double TargetQCcoef[] = {1, 0.2, 0.5, //param0
                              1, 0.7,      //param1
                              1};          //param2

      nErrorCode = LSsampInduceCorrelation(paSample,nDim,
      LSCORR_PEARSON, TargetQCnonzeros, TargetQCvarndx2,
      TargetQCvarndx1, TargetQCcoef);
     APIERRORCHECK;
   }

   // Retrieve sample points into local arrays pCIX[][]
   for (i=0; i< nDim; i++)
     LSsampGetCIPoints(paSample[i],&nSampSize,&pCIX[i]);

}
```

Automatic Sampling of Scenario Trees

As an alternative to generation of explicit sample points to be used for setting up explicit scenarios, LINDO API offers an easy to use function `LSloadSampleSizes()` to create finite scenario trees implicitly with user-specified dimensions. This is especially handy when there are several stochastic parameters and the task of explicit sampling becomes tedious. In this context, the user can specify the dimensions of a scenario tree by either of the following methods:

- **Specify the number of nodes per stage:** In this method, the user should provide an integer array of length T (number of stages in the model) and give in each position the number of nodes to be created in that stage. By default stage-0 will always one node, thus the 0^{th} index in the array will be one. Other positions in the array, corresponding to the number of nodes in stages 1,2,…T-1, may take any positive integer values. In this framework, each node represents a block realization of all the stochastic parameters in that stage and will have a conditional probability of $1/N_t$, where N_t represents the number of nodes in stage t.
- **Specify the sample size per stochastic parameter:** In this method, the user should provide an integer array of length S (the number stochastic parameters in the model), and give in each position the samples size for that stochastic parameter.

In either case, LINDO API will automatically construct a finite scenario tree with specified dimensions. The user can optionally specify the variance reduction technique with `LS_IPARAM_STOC_VARCONTROL_METHOD` parameter (the default variance reduction/control method is `LS_LATINSQUARE`). The following code snippet illustrates the first method for the Newsvendor problem (case 2) in C language.

```
{
  int   panSampleSize[]   = {1, 6, 6};

  errorcode=LSsetModelIntParameter(pModel,
                       LS_IPARAM_STOC_VARCONTROL_METHOD,
                       LS_ANTITHETIC);

  errorcode=LSloadSampleSizes(pModel,panSampleSize);

  if (errorcode !=0) { fprintf(stdout,"\nError=%d\n",errorcode);
  exit(1);}

}
```

In the Newsvendor problem under case 2, both stochastic parameters are normally distributed each belonging to a different stage. Therefore, creating N nodes per stage has the same effect as creating N samples per stochastic parameter whenever there is a single stochastic parameter per stage.

Note: Sampling a scenario tree is not limited to stochastic parameters that follow parametric distributions. It is also possible to use sampling for models, which already have a finite scenario tree. This is especially useful when the original tree is finite but still too big to handle computationally. Consider a model with 30 stochastic parameters with two outcomes each. This will correspond to a scenario tree with 2^{30} = 1.0737e+009 scenarios. Sampling will be essential for models with scenario trees this big.

Sample SP Problems

An Investment Model to Fund College Education:

We consider a four-period investment planning model to fund college education, based on the book *Introduction to Stochastic Programming*, by J. Birge and F. Louveaux. There are two investment types at each stage, Stocks (S) and Bonds (B). The objective is to maximize the wealth (Z) at the end of period 4.

Stochastic Parameters:
R_{tk} : random return from investment type k=B,S in stage, t =1, 2, 3.

Deterministic Parameters:
Initial wealth: $55,000
Target wealth: $80,000

Decision Variables:
X_{tk}: Amount invested on investment type k=B,S in stage t, t =1, 2, 3;
Z: total wealth ($1000) at the end of period 4;
Y: amount fell short from target wealth at the end of period 4;

CORE Model:

The CORE model has the following formulation. Refer to sample application under samples/c/ex_sp_putoption directory for its representation in MPI format.

```
[ COST]        MIN = 4 * Y - Z;
[ STAGE1A]   + X1B + X1S = 55;
[ STAGE2A]   - R1B * X1B - R1S * X1S + X2B + X2S = 0;
[ STAGE3A]   - R2B * X2B - R2S * X2S + X3B + X3S = 0;
[ STAGE4A]   + R3B * X3B + R3S * X3S - Z = 0;
[ STAGE4B]   + R3B * X3B + R3S * X3S + Y >= 80;
```

TIME Structure:

The time structure of constraints, variables and stochastic parameters are as follows:

Variables	Variable Index	Stage Index
X_{1B}	0	0
X_{1S}	1	0
X_{2B}	2	1

X_{2S}	3	1
X_{3B}	4	2
X_{3S}	5	2
Z	6	3
Y	7	3
Constraints	**Constraint Index**	**Stage Index**
STAGE1A	0	0
STAGE2A	1	1
STAGE2A	2	1
STAGE3A	3	2
STAGE4A	4	3
STAGE4B	5	3
Random Parameters	**Parameter Index**	**Stage Index**
R_{1B}	0	1
R_{1S}	1	1
R_{2B}	2	2
R_{2S}	3	2
R_{3B}	4	3
R_{3S}	5	3

Refer to the sample application for the steps taken to load this time structure to LINDO API.

Stochastic Structure:

The joint distribution of investment returns remain unchanged across stages and have the following form:

Outcomes	Returns (Stocks, Bonds)	Probability
High Performance	(25%, 14%)	0.5
Low Performance	(6%, 12%)	0.5

This stochastic structure can be loaded as block realizations of R_{tk} for each stage with `LSaddDiscreteBlocks` function. This is illustrated in sample application under samples/c/ex_sp_bondstok directory on your installation directory.

Running the application yields the following first stage decisions

X_{1B} = 13.520727707
X_{1S} = 41.479272293

, with the expected value of the objective function being

E[4Y - Z] = 1.514084643

For a detailed output, see the log produced by the sample application.

An American Put-Options Model:

This is a stochastic programming version of an American Put-Option as a six period model. The holder of the option has the right to sell a specified stock at any time (the feature of American options) between now and a specified expiration date at a specified strike price. The holder makes a profit in the period of exercise if the strike price exceeds the market price of the stock at the time of sale. Wealth is invested at the risk free rate. The objective is to maximize the wealth at the end of planning horizon.

Initial Price = $100

Strike price = $99

Risk free rate = 0.04%

Stochastic Parameters:

RV_t : random return in the end of period t, for $t = 0..4$

Decision Variables:

P_t: Price of option in the beginning of period t, for $t = 0…5$

W_t: Wealth at the beginning of period t, for t = 0…5

Y_t: 1 if sold in the beginning of period t, 0 otherwise, for $t = 0…5$

CORE Model:

The CORE model has the following formulation. Refer to sample application under samples/c/ex_sp_putoption directory for its representation in MPI format.

```
[OBJ] MAX= W5 ;

[PRICE0]        P0 = 100 ;     !price at t=0;
[PRICE1] RV0 * P0 = P1   ;     !price at t=1;
[PRICE2] RV1 * P1 = P2   ;     !price at t=2;
[PRICE3] RV2 * P2 = P3   ;     !price at t=3;
[PRICE4] RV3 * P3 = P4   ;     !price at t=4;
[PRICE5] RV4 * P4 = P5   ;     !price at t=5;

[WEALTH0]            + Y0 * ( 99 - P0) = W0  ;  !wealth at t=0;
[WEALTH1] 1.04 * W0 + Y1 * ( 99 - P1) = W1  ;  !wealth at t=1;
[WEALTH2] 1.04 * W1 + Y2 * ( 99 - P2) = W2  ;  !wealth at t=2;
[WEALTH3] 1.04 * W2 + Y3 * ( 99 - P3) = W3  ;  !wealth at t=3;
[WEALTH4] 1.04 * W3 + Y4 * ( 99 - P4) = W4  ;  !wealth at t=4;
[WEALTH5] 1.04 * W4 + Y5 * ( 99 - P5) = W5  ;  !wealth at t=5;

[SellOnce] Y0 + Y1+ Y2+ Y3 + Y4 +  Y5 <= 1 ; ! sell only once;

@FREE(Wt); t=0..5;
@FREE(Pt); t=0..5;
@BIN(Yt); t=0..5;
```

TIME Structure:

The time structure of constraints, variables and stochastic parameters are as follows:

Variables	**Variable Index**	**Stage Index**
P_t	t	t=0…5
W_t	t+6	t=0…5
Y_t	t+12	t=0…5
Constraints	**Constraint Index**	**Stage Index**
$PRICE_t$	t	t=0…5
$WEALTH_t$	t+6	t=0…5
SellOnce	12	5
Random Parameters	**Parameter Index**	**Stage Index**
RV_t	t-1	t=1…5

Refer to the sample application for the steps taken to load this time structure to LINDO API.

Stochastic Structure:

The discrete independent distribution of the returns for each stage is as follows:

Stages	**Returns**	**Probabilities**
1	(-8%, 1%, 7%, 11%)	(0.25,0.25,0.25,0.25)
2	(-8%, 1%)	(0.5,0.5)
3	(7%, 11%)	(0.5,0.5)
4	(1%, 11%)	(0.5,0.5)
5	(-8%, 7%)	(0.5,0.5)

This stochastic structure can, too, be expressed with block realizations of RV_t for each stage `LSaddDiscreteBlocks` as given in sample application under samples/c/ex_sp_putoption directory on your installation directory. Note, it is also possible to use `LSaddParamDistIndep` to load this structure.

Running the application yields the following first stage decision

$$Y_0 = \quad 0 \text{ (don't sell)},$$

with the expected value of the objective function being

$$E[W_5] = 3.807665$$

For a detailed output, see the log produced by the sample application.

Appendix 8a: Correlation Specification

The LINDO API supports three different ways of computing the correlation of two random variables: Pearson correlation, Spearman rank correlation, or Kendall-tau rank correlation. To describe them, first define:

$$\overline{x} = \sum_{i=1}^{n} x_i / n; \qquad s_x = \sqrt{\sum_{i=1}^{n} (x_i - \overline{x})^2 / (n-1)};$$

Pearson correlation is computed by the formula:

$$\rho_s = \sum_{i=1}^{n} (x_i - \overline{x})(y_i - \overline{y}) / (n s_x s_y);$$

Spearman Rank correlation is computed in the same way as Pearson, except x_i and y_i are replaced by their ranks, with special adjustments when there are ties.

Kendall Tau Rank

The Kendall-tau rank correlation is calculated by the formula:

$$\rho_\tau = \sum_{i=1}^{n} \sum_{k=i+1}^{n} 2 * sign[(x_i - x_k)(y_i - y_k)] / [n(n-1)]$$

where the *sign*( ) function is either +1, 0, or -1 depending upon whether its argument is either > 0, = 0, or < 0.

The advantage of the Spearman and Kendall tau correlation coefficient is that rank correlations are non-parametric. E.g., if you compute the Kendall tau correlation for a set of uniform random variables, and then transform these uniforms into Normal random variables using monotonic increasing transformations, the Kendall tau correlation remains unchanged.

Example:

Consider the data set:

X	Y
2	1.2
1	2.3
4	3.1
3	4.1

The Pearson, Kendall tau, and Spearman correlations between *X* and *Y* are respectively: 0.4177, 0.3333, and 0.4500.

Appendix 8b: Random Number Generation

The LINDO API allows the user to specify one of six random number generators:

1) LS_RANDGEN_LINDO1: Composite of linear congruentials with a long period,(default),

2) LS_RANDGEN_LINDO2: Linear congruential (31-bit),
3) LS_RANDGEN_MERSENNE: Mersenne Twister with long period.
4) LS_RANDGEN_SYSTEM: Built-in generator based on C functions rand() and srand().
5) LS_RANDGEN_LIN1: An alternative linear congruential generator.
6) LS_RANDGEN_MULT1: A multiplicative generator.

The 31-bit linear congruential generator (LS_RANDGEN_LINDO2) uses the recursion:

$IU(t) = 742938285 * IU(t\text{-}1) \text{ MOD } 2147483647$

$U(t) = IU(t) / 2147483647.0$

This generator has a cycle length of (2^31)-1, or about 2.147*10^9.

The composite generator (LS_RANDGEN_LINDO1) uses the recursion, see L'Ecuyer et al.:

$x(t) = (1403580 * x(t\text{-}2) - 810728 * x(t\text{-}3)) \text{ mod } 4294967087;$

$y(t) = (527612 * y(t\text{-}1) - 1370589 * y(t\text{-}3)) \text{ mod } 4294944443;$

$z(t) = (x(t) - y(t)) \text{ mod } 4294967087;$

$U(t) = z(t)/4294967088 \text{ if } z(t) > 0;$

$= 4294967087/4294967088 \text{ if } z(t) = 0;$

Although this generator is slower, it has the advantages that it has a cycle length of about 2^191 = 3.14*10^57. It has been shown to have good high dimension uniformity in up to 45 dimensional hypercubes.

The univariate distributions supported are Beta, Binomial, Cauchy, Chisquare, exponential, F, Gamma, Geometric, Gumbel, Hypergeometric, Laplace, Logarithmic, Logistic, Lognormal, Negativebinomial, Normal, Pareto, Poisson, Student-t, Uniform, Weibull.

Generating internally a random number from an arbitrary distribution, e.g., Normal, Poisson, Negative binomial follow the following simple steps.

1) Generate a uniform random number in (0, 1) with one of the available generators.
2) Convert the uniform to the desired distribution via the inverse transform of the cdf (cumulative distribution function).

Appendix 8c: Variance Reduction

The LINDO API provides two methods for reducing the variance of results: Latin Hyper Cube Sampling (LHS), and Antithetic Variates (ATV). Assume we want n random variables drawn from the interval (0, 1), with all outcomes equally likely, i.e., uniformly distributed.

LHS will partition the interval (0, 1) into n intervals, each of length $1/n$, and then draw one sample uniformly from each interval. For example, if $n = 10$, you might get the following sample.

0.002773	0.279945
0.789123	0.941034
0.554321	0.837275
0.376877	0.133699
0.430992	0.672890

Notice that there is exactly one number with a fraction starting with .0, one starting with .1, etc. This is extended to arbitrary distributions so that there is exactly one number drawn from the lowest $1/n$ fractile, one from the second lowest fractile, etc.

ATV sampling assumes that n is an even number. Again, assuming we want n random numbers uniform in (0, 1), ATV, ATV first draws $n/2$ numbers, $xu_1, xu_2, \ldots xu_{n/2}$ uniform in (0, 1). ATV then generates the remaining $n/2$ numbers by the rule: For $k = n/2+1$, to n: $xu_k = 1 - xu_{k-n/2}$. For example, the following $n = 10$ numbers satisfy that feature:

0.002773	0.997227
0.789123	0.210877
0.554321	0.445679
0.376877	0.623123
0.430992	0.569008

Appendix 8d: The Costs of Uncertainty: EVPI and EVMU

We should always be concerned with how much uncertainty is costing us. There are three general approaches we can take in the face of uncertainty:

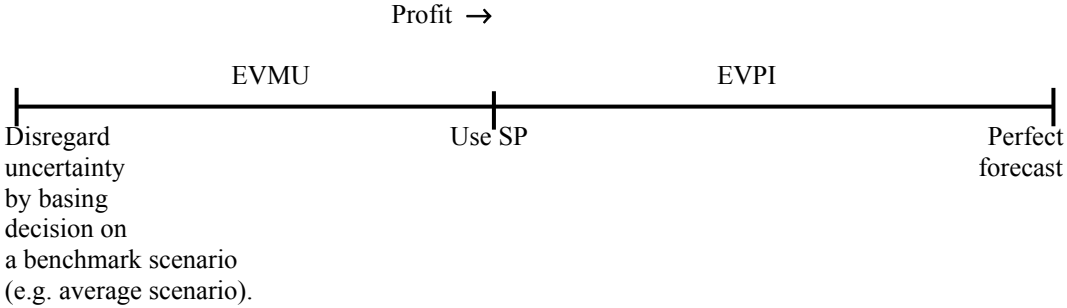
1) Disregard uncertainty. Act as if each stochastic parameter is a constant. E.g., at the beginning of each day, assume it will be partly cloudy.
2) Take uncertainty into account and prepare for it, i.e., make decisions that better take into account the possible uncertain future outcomes. E.g., Carry a small umbrella in case it is really cloudy.
3) Eliminate uncertainty. In addition to (2), do better forecasting so that uncertainty is less of an issue. E.g., subscribe to a super accurate weather forecasting service and take along a sturdy umbrella on those days when you know it will rain.

In terms of expected profit, if it costs us nothing to do the better information processing of approaches (2) and (3), then it is clear that the least profitable approach is (1), and the most profitable approach is (3).

There are two measures of the cost of uncertainty corresponding to differences in the above three:

1) EVPI (Expected Value of Perfect Information) : Expected increase in profit if we know the future in advance.
2) EVMU (Expected Value of Modeling Uncertainty) : Expected decrease in profit if we replaced each stochastic parameter by a single estimate and act as if this value is certain. EVMU is sometimes also called VSS (Value of the Stochastic Solution).

Graphically EVMU and EVPI can be described as the differences in profits for three different ways of making our decision:



Typically, the benchmark scenario is the average-scenario obtained by taking the mean of all stochastic parameters, but there may be reasons to use the median, or some other scenario. We discuss this issue later.

EVPI and EVMU Example

Consider the plant location with random demand. Each plant, if we install or keep it, has a specified capacity. For each plant customer combination there is an net revenue contribution per unit. The complete data are specified below.

```
DATA:
  PLANT = ATL  STL  CIN;  ! The 3 plants;
    CAP =  22   22   15;  ! Capacities;
  FCOST =  20   20   20 ; ! Fixed costs;
  CUST =
    CHI   SAN   NYC  MIA; ! The 4 customers;


  REV =
      8     6    7    8   ! Revenues per unit for each;
      9     7    1    1   ! combination of ;
      7     6    8    9;  ! plant & customer ;
  SCENE = 1   2    3;      ! There are 3 scenarios...;
  PWT = 0.3  0.3  0.4;     !  with probabilities...;
  DEM =
       10   10    1   1   ! Demand scenario 1;
        1    1    5   5   ! Demand scenario 2;
        2    2    3   3;  ! Demand scenario 3;
ENDDATA
```

Below we give details on the calculations.

EVPI Example Computations

If we know future only probabilistically it can be shown that the optimal policy is to open the plant in Atlanta. In this case, expected total profit = 82.40

If we know in advance that the scenario will be 1, then Expected Profit= 142.00 (Probability=0.3)
Plants to open: STL

If we know in advance that the scenario will be 2, then Expected Profit= 78.00 (Probability=0.3)
Plants to open: CIN

If we know in advance that the scenario will be 3, then Expected Profit= 57.00 (Probability=0.4)
Plants to open: CIN

So the expected Profit with Perfect Information 0.3*142 + 0.3*78 + 0.4*57 = 88.80

Recall that the Expected Profit without perfect information was 82.40.

So Expected Value of Perfect Information(EVPI)= 88.80 – 82.40 = 6.40

Notice Atlanta not optimal for any scenario!

EVMU Example Computations

If we act as if mean demand is certain...

The demand vector is:

4.1 4.1 3 3.

If we thought that the demand would be exactly ( ), then the optimal set of plants to open is CIN.

If we force the solution: CIN to be the only plant open, with all other plants closed, then in the face of the actual demand distribution, the actual expected profit with this configuration= 71.7. So doing the calculations:

```
Expected Profit Modeling uncertainty    =    82.40
Expected Profit using expected values   =  - 71.70
Expected Value of Modeling Uncertainty  =    10.70
```

EVMU, When is it zero?

Can we predict when EVMU = 0?

E.g.,

Situation 1:

The price we get for our products are stochastic parameters.

Situation 2:

The demands for our products are stochastic parameters.

EVMU and EVPI, True vs. Estimatied

A fine point: If the true number of scenarios is large, or infinite, and we use sampling, then the values for EVPI and EVMU reported are estimates rather than true values.

EVMU: Choosing the Benchmark

EVMU is the expected opportunity cost of using a policy based on a single outcome forecast of the future, relative to using a policy that is optimal taking into account the distribution of possible future outcomes. The EVMU provides a measure of how much it is costing the decision maker to not properly take into account uncertainty. Four possible single outcome forecasts come to mind. Each has its own problems. Some possible single forecast choices are:

1) Choose the policy that is optimal assuming the future outcome is always the mean outcome. This is the default benchmark scenario used in EVMU computations with LINDO API.
2) Choose the policy that is optimal assuming the future outcome is always the median outcome.
3) Choose the policy that is optimal assuming the future outcome is always the most likely outcome.
4) The user arbitrarily specifies either a point forecast or a policy, e.g. stock enough inventory so that the probability of stock out is 0.05.

Some problems with each the user should be aware of are:

1)
   a. The mean may not be defined for certain distributions, e.g., the Cauchy, or more generally the class of fat tailed "Stable Paretian" distributions popular in finance.
   b. The mean (with a fractional value) may not make sense for discrete distributions in certain situations. E.g., We are playing Rock-Paper-Scissors or some Heads-or-Tails game, and the user models the recourse decision with IF statements or a VLOOKUP. The median and most likely do not have this problem.

2)
   a. The median is ambiguous if there are an equal number of equally likely outcomes.
   b. The median is not obviously defined for a multi-dimensional stochastic parameter/vector.

3)
   a. The most likely outcome may be ambiguous, e.g., for a uniform distribution.
   b. The most likely outcome may be a nonsensical choice for a highly skewed distribution.

E.g. the most likely outcome for an exponential distribution is 0, even though the mean may be 100.

What to do?

The following "repair" actions seem appropriate for first two cases.

1) User specifies the mean, however,
   a. the mean does not exist. The typical distributions for which the mean does not exist are symmetric, so automatically switching to the median seems reasonable.
   b. there is no feasible solution to the model when a fractional value (which is usually the case for the mean) is specified for a stochastic parameter but the model expects to be integer valued. Simply report that EVMU = $+\infty$. Alternatively, one could round the mean to the nearest value that corresponds to a draw from the true population. This is easy for a univariate distribution. Not so easy for a multivariate distribution.
   c. Theoretically, the EVMU is undefined if the original SP is infeasible. For example, suppose the user says the cost of not satisfying all demand is infinite and there is an upper bound on how much can be stocked and there happens to be a possible demand greater than this upper bound. The EVMU in this case is $\infty - \infty$, which is "undefined". However, LINDO API adopts $\infty - \infty = 0$, implying that stochastic modeling of uncertainity did not lead to any additional benefits over using the benchmark scenario.

2) When using the median,
   a. Resolve the ambiguity by defining the median as the first outcome for which thecumulative sum of probabilities is equal to or greater than 0.5. This is the default strategy adopted by LINDO API when using the median as the benchmark scenario. A slightly fancier choice would be the outcome for which |cum_sum – 0.5| is smaller, breaking ties by choosing the larger cum_sum.
   b. For a vector of discrete stochastic parameters, assume the user has input the scenarios in a reasonable order. Sum up the probabilities of the scenarios starting with the first. Define the median scenario as the one for which | cum_sum – 0.5| is smaller.

Chapter 9: Using Callback Functions

In many instances, solving a model can be a lengthy operation. Given this, it may be useful to monitor the progress of the optimization. This is particularly true when building a comprehensive user interface. You may wish to display a window for the user that summarizes the solver's progress. This can be accomplished with a *callback function*—so named because the code calls the solver, and the solver periodically *calls back* to your supplied callback routine.

This chapter illustrates the use of callback functions in conjunction with LINDO API. In this section, the C and VB code samples presented in the previous chapter will be modified in order to incorporate a simple callback function. LINDO API also supports a special callback routine for integer models, where the routine is called every time the solver finds a new integer solution. This chapter is concluded with a brief discussion on the use of this integer programming callback function.

Specifying a Callback Function

To specify a callback function, call the *LSsetCallback*() routine before calling the *LSoptimize*() or the *LSsolveMIP*() solution routines. Using C programming conventions, the calling sequence for *LSsetCallback*() is:

```
int LSsetCallback(
   pLSmodel   pModel,
   cbFunc_t   pCallback,
   void*      pUserData
)
```

where,

pModel – is a pointer to the model object you wish to monitor with your callback routine.
pCallback – is a function pointer, which points to the callback routine you are supplying. To cancel an existing callback function, set *pCallback* to NULL. The callback function type *cbFunc_t* is defined in the *lindo.h* file.
pUserData – can point to whatever data you want. LINDO API merely passes this pointer through to your callback routine. You may then reference this pointer in your callback routine in order to access your data areas. Passing this pointer allows you to avoid the use of global data, thus allowing your application to remain thread safe.

The callback function you create must have the following interface:

```
int CALLBACKTYPE MyCallback(
   pLSmodel  pModel,
   int       nLocation,
   void*     pUserData
)
```

where,

pModel – is a pointer to the model object you passed to the solver. You will need to pass this pointer when retrieving information about the status of the solver. Details on retrieving information are discussed below.
nLocation – indicates the solver's current location. Its value is of no particular interest to your application. However, you may need to know the current location of the solver since there may be several different optimizers involved while solving a specific problem. For instance, in solving a nonlinear mixed-integer model, the solver will deploy both the nonlinear and MIP optimizer, and at consecutive callback times the solver may be at another location.
pUserData – is the pointer to your data area, which you originally passed to the *LSsetCallback*() routine. This can be referenced here to gain access to your data.
Return Value – is the return value of the callback function, which is used to indicate whether the solver should be interrupted or continue processing the model. To interrupt the solver, return a –1. To have the solver continue, return a 0.

The *CALLBACKTYPE* macro is declared in the *lindo.h* header file. Under Windows, *CALLBACKTYPE* is simply defined as "_stdcall_", which forces the callback function to use the standard function calling protocol rather than the C-style "cdecl" protocol. VB users don't need to worry about this aspect of the callback function because VB automatically uses standard calls.

The callback function will be called on a regular basis by the LINDO API solver. The frequency of callbacks can be controlled through the parameter *LS_DPARAM_CALLBACKFREQ*, which may be set through calls to *LSsetEnvDouParameter*(). The default value for this parameter is .5, which means the solver will callback the code approximately once every ½ second.

Once the callback function has control, you will most likely want to retrieve information regarding the solver's status. The function *LSgetCallbackInfo*() is designed for this purpose. Note that inside the callback routine, any queries directed to LINDO API must be done through *LSgetCallbackInfo*(). Other LINDO API query routines may not return valid results while the solver is invoked. Here is the interface for *LSgetCallbackInfo*():

```
int LSgetCallbackInfo(
   pLSmodel pModel,
   int nLocation,
   int nQuery,
   void* pResult
)
```

where,

pModel – is the model object pointer that was passed to your callback routine.
nLocation – is the integer value indicating the solver's current location that was passed to the callback routine. The following callback locations are possible:

Solver Location	Names
Primal Simplex Optimizer	LSLOC_PRIMAL
Dual Simplex Optimizer	LSLOC_DUAL
Barrier Optimizer	LSLOC_BARRIER
Barrier Crossover Process	LSLOC_CROSSOVER
MIP Optimizer	LSLOC_MIP
Standard Nonlinear Optimizer	LSLOC_CONOPT
Multistart Nonlinear Optimizer at a Local Optimal	LSLOC_LOCAL_OPT
Start of Instruction list-based model generation	LSLOC_GEN_START
Processing Instruction list-based model generation	LSLOC_GEN_PROCESSING
End of Instruction list-based model generation	LSLOC_GEN_END
Global Optimizer	LSLOC_GOP
Multistart Solver	LSLOC_MSW
Function Evaluation	LSLOC_FUNC_CALC
Presolver	LSLOC_PRESOLVE
Exiting the Solver	LSLOC_EXIT_SOLVER

nQuery – is the code for the object whose value you wish to retrieve. The possible values for this argument are listed in *Callback Management Routines* section under LSgetCallbackInfo description on page 227.

pResult – is a pointer to the memory location where LINDO API should store the value for the requested object. Be sure to allocate enough space for the object. Objects whose names begin with "LS_I" (e.g., LS_IINFO_SIM_ITER) return an integer quantity, while those beginning with "LS_D" return a double precision quantity.

Return Value – is the function's return value, which will be 1 if the parameter code was not recognized, else 0.

A Callback Example Using C

In this section, we will illustrate the use of a callback function written in C. The sample C application in Chapter 3, *Solving Linear Programs*, has been modified, so that it now incorporates a simple callback function. If you are not familiar with the C example in Chapter 3, *Solving Linear Programs*, review it now before proceeding with this example. The code for this example is contained in the file \lindoapi\samples\c\samp2\samp2.c. The contents of this file are reproduced below. Changes added to the file presented in Chapter 3, *Solving Linear Programs,* are displayed in bold type:

```
/* samp2.c
A C programming example of interfacing with the
  LINDO API that employs a callback function.

  The problem:
     MAX = 20 * A + 30 * C
     S.T.       A +  2 * C  <= 120
                A           <=  60
                          C  <=  50
   Solving such a problem with the LINDO API involves
   the following steps:
      1. Create a LINDO environment.
      2. Create a model in the environment.
      3. Specify the model.
      4. Perform the optimization.
      5. Retrieve the solution.
      6. Delete the LINDO environment.
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
/* LINDO API header file */
#include "lindo.h"

/* Define a macro to declare variables for
    error checking */
#define APIERRORSETUP
   int nErrorCode;
   char cErrorMessage[LS_MAX_ERROR_MESSAGE_LENGTH]

/* Define a macro to do our error checking */
#define APIERRORCHECK
   if (nErrorCode)
   {
      if ( pEnv)
      {
         LSgetErrorMessage( pEnv, nErrorCode,
          cErrorMessage);
         printf("Errorcode=%d:  %s\n", nErrorCode,
          cErrorMessage);
      } else {
         printf( "Fatal Error\n");
      }
      exit(1);
   }
```

```
/* A callback function that will be called by the LINDO
    solver */
int CALLBACKTYPE MyCallback( pLSmodel pMod, int nLocation,
 void* pMyData)
{
/* Display the string we passed to LSsetCallback() */
   printf( "In MyCallback: %s\n", pMyData);
/* Display current iteration count and objective value */
   {
      int nIter;
      double dObj;
      LSgetCallbackInfo( pMod, nLocation, LS_IINFO_SIM_ITER,
       &nIter);
      LSgetCallbackInfo( pMod, nLocation, LS_DINFO_POBJ,
       &dObj);
      printf( "In MyCallback, Iters, Obj: %d %g\n",
       nIter, dObj);
   }
   return( 0);
}

/* main entry point */
int main()
{
   APIERRORSETUP;
   int i, j;
   char strbuffer[255];
   char MY_LICENSE_KEY[1024];
/* Number of constraints */
   int nM = 3;
/* Number of variables */
   int nN = 2;
/* declare an instance of the LINDO environment object */
   pLSenv pEnv;
/* declare an instance of the LINDO model object */
   pLSmodel pModel;
/* >>> Step 1 <<< Create a LINDO environment. */
   nErrorCode = LSloadLicenseString(
               "../../../license/lndapi60.lic",MY_LICENSE_KEY);
   APIERRORCHECK;
   pEnv = LScreateEnv ( &nErrorCode, MY_LICENSE_KEY);
   if ( nErrorCode == LSERR_NO_VALID_LICENSE)
   {
      printf( "Invalid License Key!\n");
      exit( 1);
   }
   APIERRORCHECK;
/* >>> Step 2 <<< Create a model in the environment. */
   pModel = LScreateModel ( pEnv, &nErrorCode);
   APIERRORCHECK;
   {
/* >>> Step 3 <<< Specify the model.
 To specify our model, we make a call to LSloadLPData,
  passing it:
 - A pointer to the model which we are specifying(pModel)
 - The number of constraints in the model
```

```
 - The number of variables in the model
 - The direction of the optimization (i.e. minimize or
 -  maximize)
 - The value of the constant term in the objective (may be zero)
 - The coefficients of the objective function
 - The right-hand sides of the constraints
 - The types of the constraints
 - The number of nonzeros in the constraint matrix
 - The indices of the first nonzero in each column
 - The length of each column
 - The nonzero coefficients
 - The row indices of the nonzero coefficients
 - Simple upper and lower bounds on the variables
*/
/* The direction of optimization */
      int nDir = LS_MAX;
/* The objective's constant term */
      double dObjConst = 0.;
/* The coefficients of the objective function */
      double adC[2] = { 20., 30.};
/* The right-hand sides of the constraints */
      double adB[3] = { 120., 60., 50.};
/* The constraint types */
      char acConTypes[3] = {'L', 'L', 'L'};
/* The number of nonzeros in the constraint matrix */
      int nNZ = 4;
/* The indices of the first nonzero in each column */
      int anBegCol[3] = { 0, 2, nNZ};
/* The length of each column.  Since we aren't leaving
    any blanks in our matrix, we can set this to NULL */
      int *pnLenCol = NULL;
/* The nonzero coefficients */
      double adA[4] = { 1., 1., 2., 1.};
/* The row indices of the nonzero coefficients */
      int anRowX[4] = { 0, 1, 0, 2};
/* Simple upper and lower bounds on the variables.
    By default, all variables have a lower bound of zero
    and an upper bound of infinity.  Therefore pass NULL
    pointers in order to use these default values. */
      double *pdLower = NULL, *pdUpper = NULL;
/* Variable and constraint names */
      char **paszVarnames, **paszConnames;
      char *pszTitle = NULL, *pszObjname = NULL, *pszRhsname = NULL,
        *pszRngname = NULL, *pszBndname = NULL;
      paszConnames = (char **) malloc(nM*sizeof(char *));
      for (i=0; i < nM; i++)
      {
        paszConnames[i] = (char *) malloc(255*sizeof(char));
        sprintf(strbuffer,"CON%02d",i);
        strcpy(paszConnames[i],strbuffer);
      }
      paszVarnames = (char **) malloc(nN*sizeof(char *));
      for (j=0; j < nN; j++)
      {
        paszVarnames[j] = (char *) malloc(255*sizeof(char));
        sprintf(strbuffer,"VAR%02d",j);
```

```
        strcpy(paszVarnames[j],strbuffer);
      }
/* We have now assembled a full description of the model.
    We pass this information to LSloadLPData with the
    following call. */
      nErrorCode = LSloadLPData( pModel, nM, nN, nDir,
       dObjConst, adC, adB, acConTypes, nNZ, anBegCol,
       pnLenCol, adA, anRowX, pdLower, pdUpper);
      APIERRORCHECK;
/* Load name data */
      nErrorCode = LSloadNameData(pModel, pszTitle,
        pszObjname, pszRhsname, pszRngname,pszBndname,
        paszConnames, paszVarnames);
   }
   {
 /* Establish the callback function */
      char* pMyData = "My string!";
      nErrorCode = LSsetCallback( pModel,
       (cbFunc_t) MyCallback, pMyData);
      APIERRORCHECK;
 /* >>> Step 4 <<< Perform the optimization */
      nErrorCode = LSoptimize( pModel,
       LS_METHOD_PSIMPLEX, NULL);
      APIERRORCHECK;
   }
   {
 /* >>> Step 5 <<< Retrieve the solution */
      double adX[ 2], adY[3],dObj;
 /* Get the value of the objective */
      nErrorCode = LSgetInfo( pModel, LS_DINFO_POBJ, &dObj) ;
      APIERRORCHECK;
      printf( "Objective Value = %g\n", dObj);
 /* Get the primal and dual values */
      nErrorCode = LSgetPrimalSolution ( pModel, adX);
      APIERRORCHECK;
      nErrorCode = LSgetDualSolution ( pModel, adY);
      APIERRORCHECK;
      printf ("Primal values:\n");
      for (j = 0; j < nN; j++)
      {
        LSgetVariableNamej(pModel,j,strbuffer);
        printf( "%s = %g\n", strbuffer, adX[j]);
      }
      printf ("\n");
      printf ("Dual values:\n");
      for (i = 0; i < nM; i++)
      {
        LSgetConstraintNamei(pModel,i,strbuffer);
        printf( "%s = %g\n", strbuffer, adY[i]);
      }
   }
 /* >>> Step 6 <<< Delete the LINDO environment */
   LSdeleteModel( &pModel);
   LSdeleteEnv( &pEnv);

  /* Wait until user presses the Enter key */
```

```
   printf("Press <Enter> ...");
   getchar();
}
```

\lindoapi\samples\c\samp2\samp2.c

There were two primary changes made to incorporate the callback function. The first change involved including the callback function with the following code:

```
/* A callback function that will be called by the LINDO
    solver */
int CALLBACKTYPE MyCallback( pLSmodel pMod, int nLocation,
 void* pMyData)
{
/* Display the string we passed to LSsetCallback() */
   printf( "In MyCallback: %s\n", pMyData);
/* Display current iteration count and objective value */
   {
      int nIter;
      double dObj;
      LSgetCallbackInfo( pMod, nLocation, LS_IINFO_SIM_ITER,
       &nIter);
      LSgetCallbackInfo( pMod, nLocation, LS_DINFO_POBJ,
       &dObj);
      printf( "In MyCallback, Iters, Obj: %d %g\n",
       nIter, dObj);
   }
   return( 0);
}
```

Your callback function must have the exact same interface as presented here. If the interface is different, then the application will in all likelihood crash once the LINDO API solver is called.

This particular callback function displays the string that was passed when it was declared with a call to *LSsetCallback*(). This pointer can be used to point to whatever data structure you'd like access to in the callback function. Use of the passed pointer allows you to avoid using global data. The callback function then makes two calls to *LSgetCallbackInfo*() to retrieve the current iteration count and objective value from the solver. These two values are then written to the standard output device.

You can build this application using the Microsoft C/C++ *nmake* utility in conjunction with the *makefile.win* file included in the same directory as the source. Refer to the discussion of the C example in Chapter 3, *Solving Linear Programs,* for detailed build instructions.

When this application is run, the following will be displayed on the screen:

```
C:\lindoapi\samples\c\samp2>samp2
In MyCallback: My string!
In MyCallback, Iters, Obj: 2 2100
In MyCallback: My string!
In MyCallback, Iters, Obj: 3 2100
In MyCallback: My string!
In MyCallback, Iters, Obj: 3 2100
Objective Value = 2100
Primal values = 60 30
```

Because this is a relatively small model, the callback function only gets called three times. Larger models will receive many callbacks from the solver.

A Callback Example Using Visual Basic

This section will illustrate the use of a callback function written in Visual Basic. The sample VB application in Chapter 3, *Solving Linear Programs*, has been modified, so that it now incorporates a simple callback function. If you are not familiar with the VB example in Chapter 3, *Solving Linear Programs*, you should review it now before proceeding with this example.

If you are using Visual Basic 5 or later, a callback function can be implemented. The ability to use a callback function relies on the Visual Basic *AddressOf* operator, which can return the address of a function. This operator does not exist in Visual Basic 4 or earlier, nor does it exist in Visual Basic for Applications.

Your VB callback functions *must be placed within standard VB modules*. If you place your callback function in a form or class module, LINDO API will not be able to callback correctly.

The code for this example is contained in the files *\lindoapi\samples\vb\samp2\samplevb.frm* and *\lindoapi\samples\vb\samp2\callback.bas*.

The following two lines in bold type were added to *samplevb.frm* presented in Chapter 3, *Solving Linear Programs,* to identify the callback function to LINDO API:

```
                .
                .
                .
errorcode = LSloadLPData(prob, m, n, LS_MAX, 0, _
 c(0), b(0), con_type, nz, Abegcol(0), ByVal 0, _
 Acoef(0), Arowndx(0), ByVal 0, ByVal 0)
Call CheckErr(env, errorcode)
'Establish the callback function
errorcode = LSsetCallback(prob, AddressOf MyCallback, ByVal 0)
'>>> Step 4 <<<:  Perform the optimization.
errorcode = LSoptimize(prob, LS_METHOD_PSIMPLEX, ByVal 0)
Call CheckErr(env, errorcode)
                .
                .
                .
```

Additions to samplevb.frm

Note that the *AddressOf* operator is used to pass the address of our callback function to *LSsetCallback*(). The callback function, MyCallback, was placed in a separate file, so it could be included as a standard module. Placing the callback function in *samplevb.frm* with the rest of the code would not have worked because *samplevb.frm* is a form module. As mentioned above, callback functions *must* be placed in standard modules.

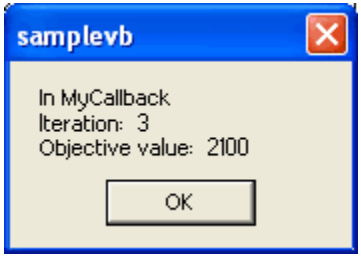
The code for MyCallback may be found in *callback.bas*, which is displayed below.

```
Public Function MyCallback(ByVal model As Long, _
 ByVal loc As Long, ByRef myData As Long) As Long
   Dim it As Long
   Dim ob As Double
   Call LSgetCallbackInfo(model, loc, LS_IINFO_SIM_ITER, it)
   Call LSgetCallbackInfo(model, loc, LS_DINFO_POBJ, ob)
   MsgBox "In MyCallback" & vbCrLf & "Iteration: " _
    & it & vbCrLf & "Objective value:  " & ob
   MyCallback = 0
End Function
```

\lindoapi\samples\vb\samp2\callback.bas

This file was included to the project by issuing the *Project|Add Module* command in Visual Basic. As with the previous C example, this callback function makes two calls to the LINDO API routine *LSgetCallbackInfo*() to retrieve the current iteration number and the objective value. The callback function returns a 0 to indicate the solver is to continue. Alternatively, a –1 may be returned to interrupt the solver.

When this application is run, the callback function should display a dialog box as follows:



Integer Solution Callbacks

In addition to the standard callback routine discussed above, LINDO API also has the ability to callback your code each time a new integer solution is found. Among other things, this will allow you to keep users of your application posted on the current best integer solution found so far. Given that large integer models can take quite some time to solve, you may want to use the callback function's ability to interrupt the solver. When LINDO API is interrupted on an integer model, it will restore the best integer solution before returning to your code. The incumbent solution may then be retrieved using normal means.

The technique for setting up your MIP callback function should look familiar because it is very similar to the technique used above for setting up a standard callback function. To set up your MIP callback, you pass its address to *LSsetMIPCallback*(). Using C programming conventions, the calling sequence for *LSsetMIPCallback*() is:

```
void LSsetMIPCallback(
   pLSmodel pModel,
   MIP_callback_t pMIPCallback,
   void* pUserData
)
```

where,

pModel – is a pointer to the model object you wish to monitor with your callback routine.
pMIPCallback – is a function pointer, which points to the callback routine being supplied. To cancel an existing callback function, set *pMIPCallback* to NULL. The *MIP_callback_t* function type is defined in the *lindo.h* header file.
pUserData – can point to any data desired. LINDO merely passes this pointer through to the callback routine. This pointer can then be referenced in the callback routine in order to access data areas. Passing this pointer avoids the use of global data, thus allowing the application to remain thread safe.

The MIP callback function created is somewhat different from the standard callback interface and must be declared as follows:

```
int CALLBACKTYPE MyMIPCallback(
   pLSModel  pModel,
   void*     pUserData,
   double    dObjective,
   double*   dPrimals
)
```

where,

pModel – is a pointer to the model object passed to the solver. This pointer will need to be passed to the solver when retrieving information about the status of the solver. Details on retrieving information are discussed below.
pUserData – is the pointer to the data area, which was originally passed to the *LSsetMIPCallback*() routine. It can be referenced here to gain access to the data.
dObjective – contains the objective value for the incumbent solution.
dPrimals – is a pointer to a double precision array containing the values of all the variables at the incumbent solution point.
Return Value – is the return value of the MIP callback function, which is presently not used and is reserved for future use. For now, this should always return a 0 value.

Once your MIP callback function has control, additional information regarding the solver's status may be retrieved. In addition to calling *LSgetCallbackInfo*() as was done from the standard callback function, *LSgetMIPCallbackInfo*() may also be called. This will return solver status information pertinent to MIP models. Here is the interface for *LSgetMIPCallbackInfo*():

```
int LSgetMIPCallbackInfo(
   pLSmodel  pModel,
   int       nQuery,
   void*     pResult
)
```

where,

pModel – is the model object pointer that was passed to the MIP callback routine.
nQuery – is the code for the object whose value to retrieve. The following objects may be retrieved:

Solver Data	**Data Type**	**Name**
Simplex iteration count	int	LS_IINFO_MIP_SIM_ITER
Barrier iteration count	int	LS_IINFO_MIP_BAR_ITER
Nonlinear iteration count	int	LS_IINFO_MIP_NLP_ITER
Objective bound	double	LS_DINFO_MIP_BESTBOUND
Branch count	int	LS_IINFO_MIP_BRANCHCOUNT
Active node count	int	LS_IINFO_MIP_ACTIVENODES
Number of relaxed problems solved	int	LS_IINFO_MIP_LPCOUNT
Returns true if an integer solution was just found.	int	LS_IINFO_MIP_NEWIPSOL
How the last integer solution was found.	int	LS_IINFO_MIP_LTYPE
Optimal objective value	double	LS_DINFO_MIP_OBJ
Solver status	int	LS_IINFO_MIP_STATUS
Objective value in the last branch solved	double	LS_DINFO_MIP_SOLOBJVAL_LAST_BRANCH
Solver status in the last branch solved	int	LS_IINFO_MIP_SOLSTATUS_LAST_BRANCH

pResult – is a pointer to the memory location LINDO API should store the value for the requested object. Be sure to allocate enough space for the object. Objects whose names begin with "LS_I" (e.g., LS_IINFO_MIP_SIM_ITER) return an integer quantity, while those beginning with "LS_D" return a double precision quantity.
Return Value – is the function's return value, which will be 1 if the parameter code was not recognized, else 0.

The mechanics of adding a MIP callback to your application are identical to what was done in the examples at the beginning of the chapter where a standard callback function was added. Users interested in adding MIP callbacks should review the next chapter to become familiar with integer modeling with LINDO API. The final section in the next chapter will direct you to specific examples that include MIP callbacks.

Chapter 10: Analyzing Models and Solutions

Sometimes after solving an optimization problem, it may be desired to get additional information beyond the standard primal and dual values of the solution. Here, two situations are considered:

1. We are unsure about the input values used. The dual prices tell us how sensitive the solution is to small changes in the input values. Over what ranges can inputs be changed without causing major changes in the solution (i.e., causing the dual prices to change)?
2. The solution was surprising. In particular, the model was infeasible or unbounded. What might be the cause of this infeasibility or unboundedness?

Sensitivity and Range Analysis of an LP

LINDO API provides three function calls that allow users to examine the sensitivity of the optimal solution of an LP to changes in model input such as right-hand side values of constraints or objective function coefficients of variables. These tools can be useful in responding better to the solution produced when model data are subject to uncertainty including, measurement errors, lack of information, and poor or partial interpretation of prices and resources.

The three function calls are:

- LSgetConstraintRanges (pLSmodel prob, double *rhsdec, double *rhsinc);
- LSgetObjectiveRanges (pLSmodel prob, double *objdec, double *objinc);
- LSgetBoundRanges (pLSmodel prob, double *boudec, double *bouinc);

The following example LP illustrates:

```
max= 20*x0 + 30*x1 + 46*x2;
 [c0]   x0            + x2 <=  60;
 [c1]          x1     + x2 <=  50;
 [c2]   x0 + 2* x1 + 3* x2 <= 120;
```

When solved, the solution is:

```
                    Primal
Variable             Value        Reduced Cost
      X0          60.00000             0.00000
      X1          30.00000             0.00000
      X2           0.00000             4.00000

     Row    Slack or Surplus      Dual Price
      C0           0.00000             5.00000
      C1          20.00000             0.00000
      C2           0.00000            15.00000
```

If *LSgetConstraintRanges*() is called, the values in the vectors *rhsdec* and *rhsinc* will be as follows:

Constraint	rhsdec	rhsinc
C0	40.00000	60.00000
C1	20.00000	LS_INFINITY
C2	60.00000	40.00000

The interpretation of these numbers is as follows. The value in:

- *rhsinc*[*i*] is the amount by which the right-hand side (RHS) of constraint *i* can be increased without causing any change in the optimal values of the dual prices or reduced costs.
- *rhsdec*[*i*] is the amount by which the RHS of constraint *i* can be decreased without causing any change in the optimal values of the dual prices or reduced costs.

For example, the allowable decrease of 20 on constraint *C1* means the RHS of 50 could be reduced by almost 20, to say 30.001, without causing any of the reduced costs or dual prices to change from (0, 0, 4, 5, 0, 15).

These are one-side guarantees in the following sense: decreasing the RHS of *C1* by more than 20 does not mean that some of the reduced costs and dual prices must change. Similarly, these are one-at-a-time guarantees. In other words, if you change multiple RHS's by less than their range limits, there is no guarantee that the reduced costs and dual prices will not change. There is, nevertheless, a 100% rule at work. Namely, if several coefficients are changed simultaneously, such that the percentage of the ranges used up is less than 100% in total, then the original guarantee still applies. For example, if the RHS of *C0* is decreased by 10 and the RHS of *C2* is decreased by 30, then the total percentage of ranges used up is 10/40 + 30/60 = 75%. Therefore, the reduced costs and dual prices would not be affected by these simultaneous changes.

If *LSgetObjectiveRanges*() is called, the values in the vectors *objdec* and *objinc* will be as follows:

Variable	objdec	objinc
X0	4.00000	LS_INFINITY
X1	4.00000	10.00000
X2	LS_INFINITY	4.00000

The interpretation of these numbers is as follows. The value in:

- *objinc*[*j*] is the amount by which the objective coefficient of variable *j* can be increased without causing any change in the optimal values of the primal values, slacks, or surpluses.
- *objdec*[*j*] is the amount by which the objective coefficient of variable *j* can be decreased without causing any change in the optimal values of the primal values, slacks, or surpluses.

For example, the allowable increase of 10 on variable *X1* means that its objective coefficient of 30 could be increased by almost 10, to say 39.999, without causing any of the primal values, slacks, or surplus values to change.

These are one-side guarantees. In other words, increasing the objective coefficient of *X1* by more than 10 does not mean that some of the primal values, slacks, or surpluses must change. Similarly, these are one-at-a-time guarantees. If you change several objective coefficients by less than their range limits, there is no guarantee that the primal values, slacks, or surpluses will not change. The 100% rule mentioned above, however, also applies here.

The function *LSgetBoundRanges*() behaves much like *LSgetConstraintRanges*(). Bounds, such as non-negativity, are just another form of constraints. For the above example, the vectors *boudec* and *bouinc* will be as follows:

Variable	boudec	bouinc
X0	LS_INFINITY	60.00000
X1	LS_INFINITY	30.00000
X2	0	30.00000

The interpretation of these numbers is as follows. The value in:

- *bouinc*[*j*] is the amount by which the lower and upper bounds of variable *j* can be increased without causing any change in the optimal values of the reduced costs and dual prices.
- *boudec*[*j*] is the amount by which the lower and upper bounds of variable *j* can be decreased without causing any change in the optimal values of the reduced costs and dual prices.

For example, the allowable increase of 60 on variable *X0* means that its lower bound of zero could be increased by almost 60, to say 59.999, without causing any of the reduced costs or dual prices to change. The allowable increase of 30 on variable *X2* means that its lower bound of zero could be increased by almost 30. If *X2* is forced to be greater-than-or-equal-to 30, then variable *X2* would be forced out of the solution.

Diagnosis of Infeasible or Unbounded Models

LINDO API contains two diagnostic tools, *LSfindIIS*() and *LSfindIUS*(), that can help users debug infeasible or unbounded optimization models. These tools can be called after the solver reports an infeasible or unbounded status for the model. *LSfindIIS*() finds an irreducible infeasible set (IIS) of constraints, whereas *LSfindIUS*(), finds an irreducible unbounded set (IUS) of variables. An IIS is a set of constraints that are infeasible taken together, but every strict subset is feasible. Similarly, an IUS is a set of variables that are unbounded taken together. However, if any one of these variables are fixed, then these variables are not unbounded. The IIS or IUS portion of the model will generally be much smaller than the original model. Thus, the user can track down formulation or data entry errors quickly. By isolating of the source of the errors, the user can correct the model data such as right-hand side values, objective coefficients, senses of the constraints, and column bounds.

Note: With LINDO API 4.0, debugging capabilities of *LSfindIIS*() have been extended beyond linear programs. It can now debug infeasible quadratic, conic, integer and general nonlinear models, too.

Infeasible Models

LSfindIIS() assumes that the user has recently attempted optimization on the model and the solver returned a "no feasible solution" message. For an LP, if an infeasible basis is not resident in the solver, *LSfindIIS*() cannot initiate the process to isolate an IIS. This can occur if the infeasibility is detected in the pre-solver before a basis is created, or the barrier solver has terminated without performing a basis crossover. To obtain an IIS for such cases, the pre-solve option should be turned off and the model must be optimized again.

The constraints and bounds in the IIS are further classified into two disjoint sets: a *necessary* set and a *sufficient* set. The *sufficient* set refers to a crucial subset of the IIS in the sense that removing any one of its members from the entire model renders the model feasible. Note that not all infeasible models have sufficient sets. The *necessary* set contains those constraints and bounds that are likely to contribute to the overall infeasibility of the entire model. Thus, the *necessary* set requires a correction in at least one member to make the original model feasible.

Example:

$$
\begin{aligned}
&\text{C1)} \quad x \qquad \geq 6;\\
&\text{C2)} \qquad y \geq 6;\\
&\text{C3)} \quad x + y \leq 5;\\
&\qquad x, \quad y \geq 0;
\end{aligned}
$$

The set *C2* and *C3* (as well as the non-negativity bound on *x*) are a necessary set. That is, some constraint in this set must be dropped or corrected. Otherwise, the model will continue to be infeasible. Note that *C1* and *C3* are also a necessary set. However, LINDO API will identify only one IIS set at a time. The constraint *C3* will be marked as a sufficient set. That is, dropping it will make the entire model feasible. Note that dropping *C2* will not make the entire model feasible, even though *C2* is a member of a necessary set. It follows that a constraint that is marked sufficient is a member of every possible necessary set. Thus, a constraint that has been marked as sufficient has a high probability of containing an error. In fact, if the model contains only one bad coefficient, the constraint containing it will be marked as sufficient.

To control the level of analysis when locating an IIS, one should pass the level (mode) of the analysis to LSfindIIS() as the second argument. Possible bit-mask values are:

LS_NECESSARY_ROWS= 1,
LS_NECESSARY_COLS = 2,
LS_SUFFICIENT_ROWS= 4,
LS_SUFFICIENT_COLS = 8.

For instance, to isolate only necessary and sufficient rows as the IIS, the associated level to pass to LSfindIIS() would be LS_NECESSARY_ROWS+ LS_SUFFICIENT_ROWS = 5.

Finally, the following methods are available to perform IIS search.

IIS Methods		
LS_IIS_DEFAULT	0	Use default filter in IIS analysis.
LS_IIS_DEL_FILTER	1	Use the standard deletion filter in IIS analysis.
LS_IIS_ADD_FILTER	2	Use the standard additive filter in IIS analysis (direct use is reserved for future releases).
LS_IIS_GBS_FILTER	3	Use generalized-binary-search filter in IIS analysis. This is a new method combining (1) and (2) with binary search.
LS_IIS_DFBS_FILTER	4	Use depth-first-binary-search filter in IIS analysis. This is an other method combining (1) and (2) using depth-first during binary search.
LS_IIS_FSC_FILTER	5	Use fast-scan filter in IIS analysis. This method deduces the IIS from the nonzero structure of the

		dual extreme ray and is more prone to numerical errors than others.
LS_IIS_ELS_FILTER	6	Use the standard elastic filter in IIS analysis. Not guaranteed to produce an IIS.

Prior to the analysis, the user can specify the norm that measures the infeasibilities with the following options.

Norm Options		
LS_IIS_NORM_FREE	0	Solver decides the infeasibility norm for IIS analysis.
LS_IIS_NORM_ONE	1	Solver uses L-1 norm for IIS analysis.
LS_IIS_NORM_INFINITY	2	Solver uses L-∞ norm for IIS analysis

Unbounded Linear Programs

LSfindIUS() is similar to LSfindIIS( ), except that if is used to track down the source of an unbounded solution in a linear program. This tool analyzes the model and isolates an irreducibly unbounded set (IUS) of columns. As in the infeasibility case, the IUS is partitioned into *sufficient* and *necessary* sets to indicate the role of the member columns in the unboundedness of the overall model.

The columns in the *sufficient* set are crucial in the sense that fixing any of these columns makes the overall model bounded. However, fixing the columns in the *necessary* set makes the IUS found a bounded set. There may still be some other unbounded set of columns in the model.

The dual of the earlier infeasibility bug example (shown above) is as follows.

Example:
Min $5u - 6v - 6w$;
Subject to:
$u - v \geq 4$;
$u - w \geq 4$;
$u, v, w, \geq 0$

The variables *u* and *v* constitute a necessary, or irreducible unbounded set. If no coefficients are changed in either of these columns, the model will remain unbounded. The variables *u* and *w* also constitute a necessary set.

The variable *u* constitutes a sufficient set. If you change its objective coefficient from 5 to 7, then the entire model becomes bounded.

Controling of the analysis level is done in a similar fashion as in previous section. For instance, to isolate only necessary and sufficient variables as the IUS, the associated level to pass to LSfindIUS() would be LS_NECESSARY_COLS+ LS_SUFFICIENT_COLS = 5. Currently, there is only a single method available to perform IUS analysis. Therefore, no other options are required to control the solver in analyzing unbounded models.

Note: Dualizing an unbounded LP would allow the user to deduce IUS results through an IIS analysis on the explicit dual model.

Infeasible Integer Programs

Infeasible integer programs with infeasible linear relaxations can be easily debugged as an infeasible LP using the standard *LSfindIIS*() for LPs. However, when the LP relaxation is feasible, the infeasible IP needs to be debugged explicitly. With the release of LINDO API 4.0, *LSfindIIS*() is also able to debug infeasible IPs. In the current implementation, variable bounds and integrality restrictions are left out of the analysis, and only structural constraints are considered. The constraints in the IIS are classified into necessary and sufficient sets just as in LP debugging.

Infeasible Nonlinear Programs

Recent enhancements in LSfindIIS() also make debugging of infeasible nonlinear models possible. Although, it is generally more difficult to determine the source of infeasibility in NLPs, LSfindIIS() performs reasonably well on a wide class of nonlinear models, particularly on quadratic and second-order-cone models. For general nonlinear models, the performance generally dependents on factors like (i) model scaling, (ii) infeasibility tolerance settings, (iii) presence of mathematical errors (e.g. log(.) of negative numbers), (iv) numerical errors (e.g. exp(.) of large numbers), (v) the initial solution selected, and (vi) convexity. For cases when it is difficult (or even impossible) to determine the feasibility status of an NLP in practical run-times, the diagnosis could lead to the isolation of a *Minimally Intractable Subsystem* (MIS), which is a small subset of the original constraint set that contributes to the intractability of the original NLP.

An Example for Debugging an Infeasible Linear Program

In this section, an application in Visual C++ 6.0 will be built that reads an infeasible linear program from an MPS file and then debugs it using LINDO API's analyze routines. A complete version of this project may be found in \lindoapi\samples\c\ex_iis.c.

```
/*
###############################################################
#                       LINDO-API
#                    Sample Programs
#
#                  Copyright (c) 2007 by LINDO Systems, Inc
#
#         LINDO Systems, Inc.           312.988.7422
#         1415 North Dayton St.         info@lindo.com
#         Chicago, IL 60622             http://www.lindo.com
###############################################################
  File   : ex_iis.c
  Purpose: Analyze an infeasible (unbounded) LP to isolate the
  constraints (variables) causing the infeasibility (unboundedness)
  of the model.
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
/* LINDO API header file */
#include "lindo.h"
```

```
/* Define a macro to declare variables for
    error checking */
#define APIERRORSETUP
   int nErrorCode;
   char cErrorMessage[LS_MAX_ERROR_MESSAGE_LENGTH]

/* Define a macro to do our error checking */
#define APIERRORCHECK
   if (nErrorCode)
   {
      if ( pEnv)
      {
         LSgetErrorMessage( pEnv, nErrorCode,
          cErrorMessage);
         printf("Errorcode=%d:  %s\n", nErrorCode,
          cErrorMessage);
      } else {
         printf( "Fatal Error\n");
      }
      exit(1);
   }

int CALLBACKTYPE MyCallback( pLSmodel pMod, int nLocation,
 void* pMyData)
{
   int *status = (int *) pMyData;
/* Display current iteration count and objective value */
   {
      int nIter,nNec=0,nSuf=0,
         nErr,nErr1,nErr2;
      double dObj, dInf;
nErr=LSgetCallbackInfo(pMod,nLocation,LS_IINFO_SIM_ITER,&nIter);
      nErr=LSgetCallbackInfo(pMod,nLocation,LS_DINFO_POBJ,&dObj);
      nErr=LSgetCallbackInfo(pMod,nLocation,LS_DINFO_PINFEAS,&dInf);
      if (status && *status == LS_STATUS_INFEASIBLE)
      {
nErr1=LSgetCallbackInfo(pMod,nLocation,LS_IINFO_NUM_IIS_ROWS,&nNec);
nErr2=LSgetCallbackInfo(pMod,nLocation,LS_IINFO_NUM_SUF_ROWS,&nSuf);
      }
      else if (status && *status == LS_STATUS_UNBOUNDED)
      {
nErr1=LSgetCallbackInfo(pMod,nLocation,LS_IINFO_NUM_IUS_COLS,&nNec);
nErr2=LSgetCallbackInfo(pMod,nLocation,LS_IINFO_NUM_SUF_COLS,&nSuf);
      }
      printf( "@MyCallback %8d, %8d, %16g, %16g, %8d (%d)\n",
        nLocation,nIter,dObj,dInf,nNec,nSuf);
   }
   return( 0);
}

static void CALLTYPE print_line_log(pLSmodel pModel, char *line, void
*userdata)
{
  if (line)
  {
    printf("\n%s",line);
```

```
    } /*if*/
  } /*print_line*/

  /* main entry point */
  int main(int argc, char **argv)
  {
     APIERRORSETUP;
     /* model data objects */
     int n; /* number of variables */
     int m; /* number of constraints */
     int solstatus;/*solution status (see lindo.h for possible
  values)*/
     int prep_level;
     char *mpsfile = NULL;
     char MY_LICENSE_KEY[1024];

     /* IIS related data objects */
     int nLevel,   /* level of analysis */
         nSuf_r,   /* number of sufficient rows     */
         nSuf_c,   /* number of sufficient columns  */
         nIIS_r,   /* number of rows in the IIS     */
         nIIS_c;   /* number of columns in the IIS  */
     int *aiRows = NULL, /* index set of rows in the IIS     */
         *aiCols = NULL, /* index set of columns in the IIS  */
         *anBnds = NULL; /* bound type of columns in the IIS */
     int j;
     char bndtype[255], oufname[255], varname[255];

     /* declare an instance of the LINDO environment object */
     pLSenv pEnv;
     /* declare an instance of the LINDO model object */
     pLSmodel pModel;
     /*********************************************************
      * Init: Command prompt calling sequence
      *********************************************************/
     {
       char szVer[255], szBld[255];
       LSgetVersionInfo(szVer,szBld);
       printf("\nAN APPLICATION FOR ANALYZING & DEBUGGING LPs\n");
       printf("\nusing LINDO API Version %s (Built
               %s)\n\n",szVer,szBld);
     }
     if (argc == 1)
     {
       printf("\nUsage: ex_iis filename\n\n");
       goto Terminate;
     }
     else if (argc == 2)
     {
       mpsfile = argv[1];
     }
    /*********************************************************
     * Step 1: Create a LINDO environment.
     *********************************************************/
     nErrorCode =
  LSloadLicenseString("../../../license/lndapi60.lic",MY_LICENSE_KEY);
```

```
   APIERRORCHECK;
   pEnv = LScreateEnv ( &nErrorCode, MY_LICENSE_KEY);
   if ( nErrorCode == LSERR_NO_VALID_LICENSE)
   {
      printf( "Invalid License Key!\n");
      exit( 1);
   }
   APIERRORCHECK;
   /*******************************************************
    * Step 2: Create a model in the environment.
    *******************************************************/
   pModel = LScreateModel ( pEnv, &nErrorCode);
   APIERRORCHECK;
   /*******************************************************
    * Step 3: Read the model from an MPS file and
    *******************************************************/
   nErrorCode = LSreadMPSFile(pModel,mpsfile,LS_UNFORMATTED_MPS);
   if (nErrorCode != LSERR_NO_ERROR)
   {
     printf("\nBad MPS format... Trying LINDO format.\n");
     nErrorCode =LSreadLINDOFile(pModel,mpsfile);
     APIERRORCHECK;
     printf("\nLINDO format OK!\n\n");
   }
   else
   {
     printf("\nMPS format OK!\n\n");
   }
   nErrorCode = LSgetInfo(pModel, LS_IINFO_NUM_VARS, &n);
   APIERRORCHECK;
   nErrorCode = LSgetInfo(pModel, LS_IINFO_NUM_CONS, &m);
   APIERRORCHECK;
   /*******************************************************
    * Step 4: Set Model parameters
    *******************************************************/
   /* Turn off the LP preprocessor. This is required if the model
   is infeasible and the user wishes to debug it. */
   nErrorCode = LSgetModelIntParameter(pModel,LS_IPARAM_LP_PRELEVEL,
     &prep_level);
   APIERRORCHECK;
   if (prep_level > 0)
     printf("The LP presolver has been turned off. Solving ...\n\n");
   nErrorCode =
LSsetModelIntParameter(pModel,LS_IPARAM_LP_PRELEVEL,0);
   /* set LP solver type for optimizations (cold start) */
   nErrorCode = LSsetModelIntParameter(pModel,LS_IPARAM_IIS_TOPOPT,
     LS_METHOD_FREE);
   /* set LP solver type for reoptimizations (warm start)*/
   nErrorCode = LSsetModelIntParameter(pModel,LS_IPARAM_IIS_REOPT,
     LS_METHOD_FREE);
#if 0
   nErrorCode = LSsetCallback( pModel,(cbFunc_t) MyCallback, NULL);
   APIERRORCHECK;
   printf( "            %8s, %8s, %16s, %16s, %8s (%s)\n",
   "LOCATION","ITERS","OBJECTIVE","INFEASIBILITY","NNEC","NSUF");
#endif
```

```
   /* Install a log function to display solver's progress
   as reported by the internal solver */
   nErrorCode = LSsetModelLogfunc(pModel, (printModelLOG_t)
print_line_log, NULL);
   nErrorCode =
LSsetModelDouParameter(pModel,LS_DPARAM_CALLBACKFREQ,0.5);
   APIERRORCHECK;
   /*****************************************************
    * Step 5: Optimize the model
    *****************************************************/
   nErrorCode = LSoptimize( pModel,LS_METHOD_FREE, &solstatus);
   APIERRORCHECK;
#if 0
   /* set callback and solution status */
   nErrorCode = LSsetCallback( pModel,(cbFunc_t) MyCallback,
&solstatus);
#endif

    if (solstatus == LS_STATUS_BASIC_OPTIMAL)
    {
      printf("\tThe model is solved to optimality.\n");
    }
   /*****************************************************
    * Step 6: Debug the model if unbounded or infeasible
    *****************************************************/
    else if (solstatus == LS_STATUS_UNBOUNDED)
    {
      APIERRORCHECK;
      printf("\nThe model is unbounded.. Analyzing...\n\n");
      nLevel = LS_NECESSARY_COLS + LS_SUFFICIENT_COLS;

      /*** Step 6.1: Find IIS ***/
      nErrorCode = LSfindIUS(pModel,nLevel);
      APIERRORCHECK;

      strcpy(oufname,"findius.ltx");
      nErrorCode = LSwriteIUS(pModel,oufname);
      printf("\n\n IUS is written to %s !!\n",oufname);
    }
    else if (solstatus == LS_STATUS_INFEASIBLE)
    {
      printf("\nThe model is infeasible.. Analyzing...\n\n");
      aiRows = (int *) malloc(m*sizeof(int));
      aiCols = (int *) malloc(n*sizeof(int));
      anBnds = (int *) malloc(n*sizeof(int));

      /*** Step 6.1: Find IIS ***/
      nLevel = LS_NECESSARY_ROWS + LS_SUFFICIENT_ROWS;

      nErrorCode = LSfindIIS(pModel,nLevel);
      APIERRORCHECK;
      nErrorCode = LSgetIIS(pModel,&nSuf_r,&nIIS_r,aiRows,
                                  &nSuf_c,&nIIS_c,aiCols,anBnds);
      APIERRORCHECK;
      printf("\n\t ***  LSfindIIS Summary ***\n\n");
      printf("\t Number of Sufficient Rows = %u\n",nSuf_r);
```

```
      printf("\t Number of Sufficient Cols = %u\n",nSuf_c);
      printf("\t Number of Necessary  Rows = %u\n",nIIS_r - nSuf_r);
      printf("\t Number of Necessary  Cols = %u\n",nIIS_c - nSuf_c);
      printf("\n");

      /*** Step 6.2: Display row index sets ***/
      printf("\n IIS Rows\n");
      for (j=0; j<nIIS_r; j++)
      {
        nErrorCode = LSgetConstraintNamei(pModel,aiRows[j],varname);
        APIERRORCHECK;
        if (j<nSuf_r)
          printf("%2d] (%-8s) is"
          " in the sufficient set.\n",j,varname);
        else
          printf("%2d] (%-8s) is"
          " in the necessary set.\n",j,varname);
      }
      /*** Step 6.3: Display column index sets ***/
      printf("\n IIS Column Bounds\n");
      for (j=0; j<nIIS_c; j++)
      {
        if (anBnds > 0)
          strcpy(bndtype,"Lower");
        else
          strcpy(bndtype,"Upper");

        nErrorCode = LSgetVariableNamej(pModel,aiCols[j],varname);
        APIERRORCHECK;
        if (j<nSuf_r)
          printf("%2d] %s bound of (%-8s) is"
          " in the sufficient set.\n",j,bndtype,varname);
        else
          printf("%2d] %s bound of (%-8s) is"
          " in the necessary set.\n",j,bndtype,varname);
      }
      strcpy(oufname,"findiis.ltx");
      LSwriteIIS(pModel,oufname);
      printf("\n\n IIS is written to %s !!\n",oufname);

      free(aiRows);
      free(aiCols);
      free(anBnds);
    }
  /*********************************************************
   * Step 7: Terminate
   *********************************************************/
    nErrorCode = LSdeleteModel( &pModel);
    nErrorCode = LSdeleteEnv( &pEnv);
Terminate:
  /* Wait until user presses the Enter key */
   printf("Press <Enter> ...");
   getchar();
}
```

After building this application, you can run it from the DOS-prompt to debug the model in *lindoapi\samples\mps\testilp.mps*. This should produce the following summary report on your screen.

```
MPS format OK!
***  LSfindIIS Summary ***
Number of Sufficient Rows = 0
Number of Sufficient Cols = 0
Number of Necessary  Rows = 2
Number of Necessary  Cols = 2
*** Rows Section ***
0] Row  4 (ROW5    ) is in the necessary set.
1] Row  0 (ROW1    ) is in the necessary set.
*** Column Bounds Section ***
0] Lower bound of Col  1 (X2      ) is in the necessary set.
1] Lower bound of Col  2 (X3      ) is in the necessary set.
IIS is written to findiis.ltx !
```

Block Structured Models

Many large-scale linear and mixed integer problems have constraint matrices that are extremely sparse. In practice, the ratio of the number of nonzeros to the total is so small (less than 0.05 %) that the underlying model generally has a structure that could be exploited in solving the model. Such models are often seen in airline scheduling, multi-period production planning, planning under uncertainty, and other logistics problems. There are four types of possible decomposition schemes for a constraint matrix.

Independent Block Structure

In this type of decomposition, the underlying model has a constraint matrix that is totally decomposable. As illustrated in Figure 9.1, this implies that the blocks forming the constraint matrix are independent from each other. Each block can be associated to a sub-problem that can be solved independently. An optimal solution to the overall problem can then be obtained by taking the union of the solutions to the sub-problems. A hypothetical case would be the minimization of operating costs of a company who owns three plants, which do not share any resources. The company can make the decisions pertaining to each plant independently.

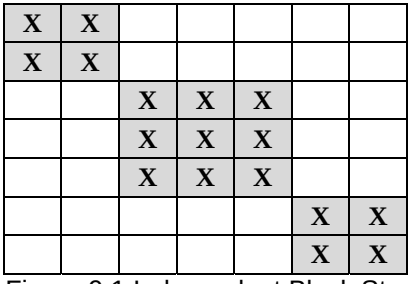


Figure 9.1 Independent Block Structure

Block Angular Structure with Linking Rows

In this type of decomposition, the blocks forming the constraint matrix are linked by a number of constraints (rows) as illustrated in Figure 9.2. Note that when all linking rows are eliminated from the constraint matrix, the remaining rows and columns form independent blocks. Therefore, the model is totally decomposable. It is always possible to transform a sparse matrix into one that has block angular structure. However, the advantages may not be available in the presence of many linking rows. Building on the hypothetical example described above, this structure can be associated to the case when there are a small number of resources that are common to all plants. In this case, the decisions involve optimal splitting of these resources among the plants efficiently.

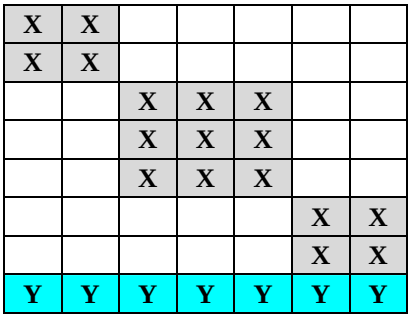


Figure 9.2 Block Angular Structure

Dual Angular Structure with Linking Columns

In this type of decomposition, the blocks forming the constraint matrix are linked by a number of variables (columns) as illustrated in Figure 9.3. This structure has a primal-dual relationship with the Block Angular Structure described above. Again, for our hypothetical plant example, a structure of this form can be associated to the case when there are a few variable outside factors that effect all plants.

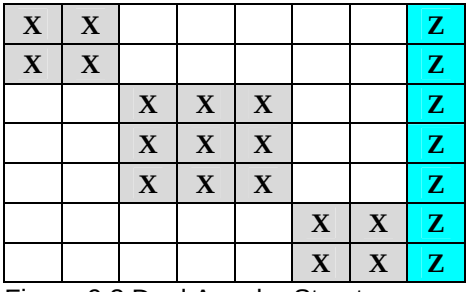


Figure 9.3 Dual Angular Structure.

Block and Dual Angular Structures

This is the most general form of decomposition where the blocks forming the constraint matrix have both linking rows and columns as illustrated in Figure 9.4. The decisions involved for the hypothetical plant example now include both resource sharing and external factors.

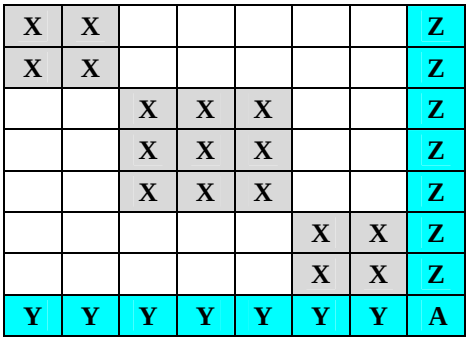


Figure 9.4 Block and Dual Angular Structure

Determining Total Decomposition Structures

Given a linear or mixed-integer program, the user can determine whether the underlying constraint matrix is totally decomposable by making a call to the *LSfindBlockStructure*() routine. If total decomposition is detected, the routine will return (i) the total number of independent blocks, (ii) an integer array indicating assignments of the constraints to the blocks, and (iii) an integer array indicating the assignments of the variables to the blocks. The following piece of C code demonstrates how *LSfindBlockStructure*() can be used to check if a model that has 100 constraints and 200 variables is totally decomposable:

```
{
   pLSmodel model;
   int nblock, type, err;
   int rblock[100], cblock[200];
   :
   :
   type = LS_LINK_BLOCKS_NONE; // try total decomposition
   err = LSfindBlockStructure(model, nblock, type);
   err = LSgetBlockStructure(model, &nblock, &rblock, &cblock, type);
   if (nblock > 1)
     printf(" The model has %d independent blocks\n",nblock);
   else
     printf(" The model is not totally decomposable\n");
   :
   :
}
```

On return, the k^{th} entry of the array cblock (rblock) will indicate the index of the block that the k^{th}-variable (k^{th}-constraint) belongs to.

Note: Many large scale linear (LP) and mixed integer problems (MIP) have constraint matrices that are totally decomposable into a series of independent block structures. The user adjustable parameter (LS_IPARAM_DECOMPOSITION_TYPE) can be set, so the solver checks if a model can be broken into smaller independent models. If total decomposition is possible, it will solve the independent problems sequentially to reach a solution for the original model. This may result in dramatic speed improvements.

Determining Angular Structures

If the matrix is not found to be totally decomposable, then other decomposition schemes can be pursued. For any constraint matrix, block-angular, dual-angular, or block-and-dual decompositions can always be achieved. In models with block- (dual-) angular decomposition, some of the rows (columns) will not belong to any of the blocks. Such rows (columns) are regarded as linking or coupling rows (columns), since they establish a dependence relationship among the blocks constituting the original matrix. *LSfindBlockStructure*() will label such rows (columns) with a zero value on return. The C code above can be modified as follows to use decomposition schemes other than total decomposition:

```
{
   pLSmodel model;
   int nblock, type, err, m=100, n=100;
   int rblock[100], cblock[200];
   :
   :
   // perform dual angular decomposition
   type = LS_LINK_BLOCKS_COLS;
   // specify the number of blocks to decompose the model
   nblock = 3;
   // perform decomposition
   err = LSfindBlockStructure(model, nblock, type);
   err = LSgetBlockStructure(model, &nblock, &rblock, &cblock, type);
   // print block memberships
   for (j=0; j<n; j++)
      if (cblock[j] > 0)
        printf(" Variable %d belongs to block %d\n",j,cblock[j]);
      else
        printf(" Variable %d is a linking column\n",j);

   for (i=0; i<m; i++)
      printf(" Constraint %d belongs to block %d\n",j,rblock[j]);
   :
   :
}
```

Note: In decomposing a constraint matrix into one that has a block and/or dual angular structure, the user needs to specify the number of blocks requested explicitly. This is because the matrix can be decomposed into as many blocks as possible when linking rows or columns are allowed.

Chapter 11: mxLINDO A MATLAB Interface

Introduction

MATLAB is a technical-computing and problem-solving environment that combines numerical analysis, matrix manipulation, and graphics tools in a user-friendly environment. This environment has a built-in high-level programming language that allows the development of special algorithms without much programming.

mxLINDO is a MATLAB executable (MEX-file) to establish an interface to LINDO API from within MATLAB. It provides MATLAB users direct access to several LINDO API routines for developing higher-level MATLAB functions (m-functions) to solve various kinds of optimization problems. The interface is particularly useful if you are solving very large or very difficult linear and integer programs, or implementing an optimization algorithm with MATLAB's programming language.

This release of the interface works with MATLAB Version 5.0 or later. The precompiled binary mxlindo.dll (also mxlindo.mexw32 or mxlindo.mexw64 for 32-bit and 64-bit Windows platforms for more recent versions of Matlab) can be found in the lindoapi\bin\winxx folder.

Setting up MATLAB to Interface with LINDO

Use the following instructions to establish an interface with MATLAB:

1. Edit the *C:\MATLAB\TOOLBOX\LOCAL\STARTUP.M* file that came with your MATLAB distribution using your favorite text editor. Typically, your MATLAB installation is under *C:\MATLAB*. For MATLAB Release 2009a, the default directory is R2009a. In more recent versions of MATLAB, the path may start with C:\Program Files\MATLAB\. If you do not have the *STARTUP.M* file, then create it from *STARTUPSAV.M*.
2. Append the following lines to the end of your *STARTUP.M* file to update your MATLAB environment-path. It is assumed that your LINDO API installation is under '*C:\LINDOAPI*'. If the last line in *STARTUPSAV.M* is "*load matlab.mat*", then delete that line.

```
global MY_LICENSE_FILE
MY_LICENSE_FILE = 'C:\LINDOAPI\LICENSE\LNDAPI50.LIC';
path('C:\LINDOAPI\BIN\WIN32',path);
path('C:\LINDOAPI\INCLUDE\',path);
path('C:\LINDOAPI\MATLAB\',path);
```

3. Start a MATLAB session and try the sample m-functions to use the interface.

Using the mxLINDO Interface

The quickest way of trying out the mxLINDO interface is to use one of the m-functions provided with mxLINDO. is version of the interface supports a subset of the available functions and routines in LINDO API. Here we demonstrate the LMsolve.m function supplied with mxLINDO.

Suppose, using matrix notation, we wish to solve:

$$\begin{aligned} &\text{Minimize} && c^T x \\ &\text{S.t.} && Ax \geq b \\ & && u \geq x \geq l \end{aligned}$$

Define the objects *A*, *b*, c, *l*, *u*, and *csense* in the MATLAB as in Figure 10.1.

```
» A = [
    1.0000    1.0000    1.0000    1.0000;
    0.2000    0.1000    0.4000    0.9000;
    0.1500    0.1000    0.1000    0.8000;
  -30.0000  -40.0000  -60.0000 -100.0000 ]
» b = [ 4000 3000 2000 -350000]'
» c = [ 65   42   64   110]'
» csense = 'GGGG';
» l=[]; u=[];
```

Figure 10.1

Setting *l* and *u* to empty vectors causes all lower and upper bounds to be at their default values (0 and LS_INFINITY, respectively). The sense of the constraints is stored in the string variable *csense*. To solve this LP, the following command should be issued at the MATLAB command prompt:

```
>> [x, y, s, dj, obj, solstat] = LMsolvem(A, b, c, csense, l, u)
```

As illustrated in Figure 10.2, the function returns the primal and dual solutions (*x,s)* and (*y,dj)*, the optimal objective value *obj*, and the optimization status flag *solstat*. *LSsolveM.m* may be modified in several ways to change the output returned.

```
» [x, y, s, dj, obj, solstat] = LMsolvem(A, b, c, csense, l, u)

x =

  1.0e+003 *

    0.1429
         0
    1.0000
    2.8571

y =

   66.0000
  202.8571
         0
    1.3857

s =

         0
         0
 -407.1429
         0

dj =

   -0.0000
   11.1429
   -0.0000
   -0.0000

obj =

  3.8757e+005

solstat =

     2
```

Figure 10.2

Further examples of this high-level use of mxLINDO and the LMsolveM.m furnction are given at the end of this chapter. LMsolve.m was built using low level calls that can be made from MATLAB to the LINDO API via the mxLINDO interface. The following section describes all the low level calls that are available in mxLINDO.

Calling Conventions

This version of the interface supports a subset of the available functions and routines in LINDO API. The calling conventions used to access these routines within MATLAB are quite similar to the C/C++ prototypes described above (see Chapter 2, *Function Definitions*). The main difference is that, when accessing any external routine within MATLAB, all arguments modified by the external routine (the output-list) appear as *left-hand* side (LHS) arguments, whereas the constant arguments (the input-list) appear as *right-hand* side (RHS) arguments.

For example, consider a LINDO API routine that has the following C/C++ prototype calling sequence:

```
int LSroutine(a1,a2,…,ak, z1,z2, …, zn)
```

Assume that this function retrieves (or modifies) the values for $z_1,z_2, \ldots, z_n$ using the input list $a_1,a_2,\ldots,a_k$. The calling convention mxLINDO uses to access this routine within MATLAB is:

```
>> [z1,z2, …, zn] = mxlindo('LSroutine',a1,a2,…,ak)
```

where ***mxlindo*** is the MATLAB executable function that calls LINDO API. The first input (right-hand side) argument of the *mxlindo* function is required to be a string that corresponds to the name of the LINDO API routine that the user wishes to access. Note that the subroutine names are case sensitive. The arguments $a_1,a_2,\ldots,a_k$ are the constant (RHS) arguments and $z_1,z_2, \ldots, z_n$ are the variable (LHS) arguments required by this routine. In naming RHS and LHS arguments, a dialect of the so-called Hungarian Notation is adopted. See Chapter 1, *Introduction*, to review the details of this naming convention.

mxLINDO Routines

In the following sections, we describe the calling sequence for all of the supported LINDO API routines. See Chapter 2, *Function Definitions*, above to review the standard calling conventions and their argument lists. Observe that the input and output arguments of mxLINDO follow the definitions therein with a few exceptions.

Note: All the parameter macros described in Chapter 2, *Function Definitions*, are also available from within MATLAB via the *lindo.m* script file located in lindoapi\include\ directory.

Structure Creation and Deletion Routines

In a standard C/C++ application that calls LINDO API, an environment or a model instance is referenced with a pointer. In MATLAB, we identify each environment and model with the integer cast of its pointer created during the call to *LScreateEnv*() or *LScreateModel*().

LScreateEnv()

Description:

Creates a new instance of *LSenv*, which is an environment used to maintain one or more models.

MATLAB Prototype:

```
>> [iEnv ,nStatus] = mxlindo('LScreateEnv', MY_LICENSE_KEY)
```

RHS Arguments:

Name	Description
MY_LICENSE_KEY	A string containing the license key file.

LHS Arguments:

Name	Description
iEnv	An integer cast to the instance of *LSenv* created.
nStatus	An integer error code. If successful, n*Status* will be 0 on return. A list of possible error codes may be found in Appendix A, *Error Codes*.

Remarks:

- This variable can be set by calling the *LSloadLicenseString*() function.

LScreateModel()

Description:

Creates a new instance of *LSmodel*.

MATLAB Prototype:

```
>> [iModel, nStatus] = mxlindo('LScreateModel', iEnv)
```

RHS Arguments:

Name	Description
iEnv	A user assigned integer referring to an instance of *LSenv*.

LHS Arguments:

Name	Description
iModel	An integer cast to the instance of *LSmodel* created.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A, *Error Codes*.

LSdeleteEnv()

Description:

Deletes an instance of *LSenv*.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSdeleteEnv', iEnv)
```

RHS Arguments:

Name	Description
iEnv	A user assigned integer referring to an instance of *LSenv*.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A, *Error Codes*.

LSdeleteModel()

Description:

Deletes an instance of *LSmodel*.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSdeleteModel', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A, *Error Codes*.

License Information Routines

The routine in this section allows you to read a license key from a license file (e.g. *\Lindoapi\License\lndapi60.lic*) and load it into a local string buffer (e.g., MY_LICENSE_KEY).

LSgetVersionInfo()

Description:

Returns the version and build information of the LINDO API on your system.

MATLAB Prototype:

```
>> [szVersion, szBuildDate, nStatus] = mxlindo('LSgetVersionInfo')
```

LHS Arguments:

Name	Description
szVersion	A null terminated string that keeps the version information of the LINDO API on your system.
szBuildDate	A null terminated string that keeps the build date of the LINDO API library on your system.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A, *Error Codes*.

LSloadLicenseString()

Description:

Reads the license string from the specified file in text format.

MATLAB Prototype:

```
>> [MY_LICENSE_KEY, nStatus] = mxlindo('LSloadLicenseString',
MY_LICENSE_FILE)
```

RHS Arguments:

Name	Description
MY_LICENSE_FILE	The global string containing the full name of the license key file.

LHS Arguments:

Name	Description
MY_LICENSE_KEY	A string containing the license key file.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A, *Error Codes*.

Remarks:

- *MY_LICENSE_FILE* is the string variable that keeps the name of your LINDO API license file and is loaded during startup. Please see *Lindoapi\Matlab\Readme.txt* for setup instructions.

Input-Output Routines

The routines in this section provide functionality for reading and writing model formulations to and from disk files into LINDO API.

LSreadLINDOFile()

Description:

Reads the model in LINDO (row) format from the given file and stores the problem data in the given model structure.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSreadLINDOFile', iModel, szFname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the model.
szFname	A string containing the path and name of the LINDO format file.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A, *Error Codes*.

LSreadMPIFile()

Description:

Reads the model in MPI format from the given file and stores the problem data in the given model structure.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSreadMPIFile', iModel, szFname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the model.
szFname	A string containing the path and name of the MPI format file.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSreadMPSFile()

Description:

Reads a model in MPS format from the given file into the given problem structure.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSreadMPSFile', iModel, szFname, nFormat)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the model.
szFname	A string containing the path and name of the basis file.
nFormat	An integer parameter indicating whether the MPS file is formatted or not. The parameter value should be either LS_FORMATTED_MPS or LS_UNFORMATTED_MPS

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A, *Error Codes*.

LSreadBasis()

Description:

Reads an initial basis from the given file in the specified format.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSreadBasis', iModel, szFname, nFormat)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the model.
szFname	A string containing the path and name of the basis file.
nFormat	An integer parameter indicating the format of the file to be read. Possible values are • LS_BASFILE_BIN : Binary format (default) • LS_BASFILE_MPS : MPS file format • LS_BASFILE_TXT : Space delimited text format

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSwriteDualLINDOFile()

Description:

Writes the dual of a given problem to a file in LINDO format.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSwriteDualLINDOFile', iModel, szFname,
nObjsense)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the model.
szFname	A string containing the path and name of the file to which the model should be written.
nObjsense	An integer indicating the sense of the dual objective function.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return.

	A list of possible error codes may be found in Appendix A.

LSwriteDualMPSFile()

Description:

Writes the dual of a given problem to a file in MPS format.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSwriteDualMPSFile', iModel, szFname,
nFormat, nObjsense)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the model.
szFname	A string containing the path and name of the MPS format file.
nFormat	An integer parameter indicating whether the MPS file is formatted or not.
nObjsense	An integer indicating the sense of the dual objective function.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSwriteIIS()

Description:

Writes the IIS of an infeasible LP to a file in LINDO file format.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSwriteIIS', iModel, szFname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szFname	A string containing the path and name of the file to which the solution should be written.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A, *Error Codes*.

LSwriteIUS()

Description:

Writes the IUS of an unbounded LP to a file in LINDO file format.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSwriteIUS', iModel, szFname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szFname	A string containing the path and name of the file to which the solution should be written.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A, *Error Codes*.

LSwriteLINDOFile()

Description:

Writes the given problem to a file in LINDO format. Model must be linear.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSwriteLINDOFile', iModel, szFname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the model.
szFname	A string containing the path and name of the file to which the model should be written.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSwriteLINGOFile()

Description:

Writes the given problem to a file in LINGO format. Model must be linear.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSwriteLINGOFile', iModel, szFname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* containing the model to be written to a LINGO file.
szFname	A string containing the path and name of the file to which the model should be written.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSwriteMPIFile()

Description:

Writes the given model in MPI format.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSwriteMPIFile', iModel, szFname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the model.
szFname	A string containing the path and name of the file to which the model should be written.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSwriteMPSFile()

Description:

Writes the given problem to a specified file in MPS format.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSwriteMPSFile', iModel, szFname, nFormat)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the model.
szFname	A string containing the path and name of the file to which the model should be written.
nFormat	An integer indicating the format of the file to be written.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSwriteBasis()

Description:

Reads an initial basis from the given file in the specified format.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSwriteBasis', iModel, szFname, nFormat)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the model.
szFname	A string containing the path and name of the file to which the model should be written.
nFormat	An integer parameter indicating the format of the file to be written. Possible values are • LS_BASFILE_BIN : Binary format (default) • LS_BASFILE_MPS : MPS file format • LS_BASFILE_TXT : Space delimited text format

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSwriteSolution()

Description:

Writes the LP solution to a file .

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSwriteSolution', iModel, szFname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szFname	A string containing the path and name of the file to which the solution should be written.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Error Handling Routines

The following command can be used to print the description of an error message to your screen.

LSgetErrorMessage()

Description:

Returns an error message for the given error code.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSgetErrorMessage', nErrorcode)
```

RHS Arguments:

Name	Description
nErrorcode	The error code associated with the error message for which you want a description.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetErrorRowIndex()

Description:

Retrieves the index of the row where a numeric error has occurred.

MATLAB Prototype:

```
>> [iRow, nStatus] = mxlindo('LSgetErrorRowIndex', iModel);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the problem data.

LHS Arguments:

Name	Description
iRow	An integer variable to return the row index with numeric error.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetFileError()

Description:

Provides the line number and text of the line in which an error occurred while reading or writing a file.

MATLAB Prototype:

```
>> [nLinenum, szLinetxt, nStatus] = mxlindo('LSgetFileError',
iModel);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the problem data.

LHS Arguments:

Name	Description
nLinenum	An integer that returns the line number in the I/O file where the error has occurred.
szLinetxt	A string that returns the text of the line where the error has occurred.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Parameter Setting and Retrieving Routines

The following routines are used to set and get various model and environment parameters with mxLINDO. Please refer to the parameter macro list given in Chapter 2, *Function Definitions*, for their definitions.

LSgetEnvParameter()

Description:

Retrieves a parameter for a specified environment.

MATLAB Prototype:

```
>> [dValue, nStatus] = mxlindo('LSgetEnvParameter', iEnv,
nParameter);
```

RHS Arguments:

Name	Description
iEnv	An integer referring to an instance of *LSenv*.
nParameter	An integer macro.

LHS Arguments:

Name	Description
dValue	On return, *dValue* will contain the parameter's value. The user is responsible for allocating sufficient memory to store the parameter value.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetEnvDouParameter()

Description:

Gets or retrieves a double precision parameter for a specified environment.

MATLAB Prototype:

```
>>[dVal, nStatus] = mxlindo('LSgetEnvDouParameter', iEnv, nParameter)
```

RHS Arguments:

Name	Description
iEnv	An integer referring to an instance of *LSenv*.
nParameter	An integer referring to a double precision parameter.

LHS Arguments:

Name	Description
dVal	A double precision variable. On return, *dVal* will contain the parameter's value.

nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetEnvIntParameter()

Description:

Gets or retrieves an integer parameter for a specified environment.

MATLAB Prototype:

```
>>[nVal, nStatus] = mxlindo('LSgetEnvIntParameter', iEnv, nParameter)
```

RHS Arguments:

Name	Description
iEnv	An integer referring to an instance of *LSenv*.
nParameter	An integer referring to an integer parameter.

LHS Arguments:

Name	Description
nVal	An integer variable. On return, *nVal* will contain the parameter's value.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetModelParameter()

Description:

Retrieves a parameter or status variable for a specified model.

MATLAB Prototype:

```
>> [dValue, nStatus] = mxlindo('LSgetModelParameter', iModel,
nParameter)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nParameter	An integer macro.

LHS Arguments:

Name	Description
dValue	On return, *dValue* will contain the parameter's value. The user is responsible for allocating sufficient memory to store the parameter value.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetModelDouParameter()

Description:

Gets or retrieves a double precision parameter for a specified model.

MATLAB Prototype:

```
>> [dVal, nStatus] = mxlindo('LSgetModelDouParameter', iModel,
nParameter)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nParameter	An integer referring to a double precision parameter.

LHS Arguments:

Name	Description
dVal	A double precision variable. On return, *dVal* will contain the parameter's value.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetModelIntParameter()

Description:

Gets or retrieves an integer parameter for a specified model.

MATLAB Prototype:

```
>> [nVal, nStatus] = mxlindo('LSgetModelIntParameter', iModel,
nParameter)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nParameter	An integer referring to an integer parameter.

LHS Arguments:

Name	Description
nVal	An integer variable. On return, *nVal* will contain the parameter's value.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSsetEnvParameter ()

Description:

Sets a parameter for a specified environment.

MATLAB Prototype:

```
>>[nStatus] = mxlindo('LSsetEnvParameter', iEnv, nParameter, dValue)
```

RHS Arguments:

Name	Description
iEnv	An integer referring to an instance of *LSenv*.
nParameter	An integer macro.
dValue	A variable containing the parameter's new value.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSsetEnvDouParameter()

Description:

Sets a double precision parameter for a specified environment.

MATLAB Prototype:

```
>>[nStatus] = mxlindo('LSsetEnvDouParameter', iEnv, nParameter, dVal)
```

RHS Arguments:

Name	Description
iEnv	An integer referring to an instance of *LSenv*.
nParameter	An integer referring to a double precision parameter.
dVal	A double precision variable.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSsetEnvIntParameter()

Description:

Sets an integer parameter for a specified environment.

MATLAB Prototype:

```
>>[nStatus] = mxlindo('LSsetEnvIntParameter', iEnv, nParameter, nVal)
```

RHS Arguments:

Name	Description
iEnv	An integer referring to an instance of *LSenv*.
nParameter	An integer referring to an integer parameter.
nVal	An integer variable.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSsetModelParameter()

Description:

Sets a parameter for a specified model.

MATLAB Prototype:

```
>>[nStatus] = mxlindo('LSsetModelParameter', iModel, nParameter,
dValue)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nParameter	An integer macro.
dValue	A variable containing the parameter's new value.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSsetModelDouParameter()

Description:

Sets a double precision parameter for a specified model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSsetModelDouParameter', iModel, nParameter,
dVal)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nParameter	An integer referring to a double precision parameter.
dVal	A double precision variable.

LHS Arguments:

Name	Description
nStatus	0 if successful, else one of the error codes listed in Appendix A.

LSsetModelIntParameter()

Description:

Sets an integer parameter for a specified model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSsetModelIntParameter', iModel, nParameter,
nVal)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nParameter	An integer referring to an integer parameter.
nVal	An integer variable.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSreadEnvParameter()

Description:

Reads environment parameters from a parameter file.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSreadEnvParameter', iEnv , szFname)
```

RHS Arguments:

Name	Description
iEnv	A user assigned integer referring to an instance of *LSenv*.
szFname	The name of the file from which to read the environment parameters.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSreadModelParameter()

Description:

Reads model parameters from a parameter file.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSreadModelParameter', iModel, szFname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szFname	The name of the file from which to read the model parameters.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSwriteModelParameter()

Description:

Writes model parameters to a parameter file.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSwriteModelParameter', iModel, szFname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szFname	The name of the file from which to read the model parameters.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Model Loading Routines

The routines in this section allow you to pass a model to LINDO API directly through memory. LINDO API expects the formulation to be in sparse format. In other words, only nonzero coefficients are passed. For details on sparse representation, see the section titled *Sparse Matrix Representation* in Chapter 1, *Introduction*.

Note: LINDO API uses the C-language type indexing of arrays. Therefore, when loading an index vector into LINDO API by using mxLINDO, make sure that the index set is a C based index set (i.e., zero is the base index).

LSloadConeData()

Description:

Loads quadratic cone data into a model structure.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSloadConeData', iModel , nCone ,
szConeTypes, aiConebegcone, aiConecols)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nCone	Number of cones to add.
szConeTypes	A character vector containing the type of each cone being added. Valid values for each cone are ‘Q’ and ‘R’. The length of this vector is equal to *nCone*.

aiConebegcone	An integer vector containing the index of the first variable that appears in the definition of each cone. This vector must have nCone+1 entries. The last entry will be the index of the next appended cone, assuming one was to be appended. If aiConebegcone [i] < aiConebegcone [i-1], then LSERR_ERROR_IN_INPUT is returned.
aiConecols	An integer vector containing the indices of variables representing each cone. The length of this vector is equal to aiConebegcone[nCone].

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSloadInstruct()

Description:

Loads an instruction lists into a model structure.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSloadInstruct', iModel , nCons , nObjs
,nVars , nNums , anObjSense , acConType , acVarType , anCode , nCode
, aiVars , adVals , adX0 , aiObj , anObj , aiRows , anRows , adL )
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nCons	Number of constraints in the model.
nObjs	Number of objectives in the model. Currently, only a single objective function is supported. (i.e., nObjs = 1)
nVars	Number of variables in the model.
nNums	Number of real numbers in the model.
anObjSense	An integer vector containing the indicator stating whether the objective is to be maximized or minimized. Valid values are LS_MAX or LS_MIN, respectively. The length of this vector is equal to *nObjs*.
acConType	A character vector containing the type of each constraint. Each constraint is represented by a single byte in the array. Valid values for each constraint are ‘L’, ‘E’, ‘G’, or ‘N’ for less-than-or-equal-to, equal to, great-than-or-equal-to, or neutral, respectively. The length of this vector is equal to *nCons*.
acVarType	A character vector containing the type of each variable. Valid

	values for each variable are 'C', 'B', or 'I', for continuous, binary, or general integer, respectively. The length of this vector is equal to *nVars*. This value may be '[]' on input, in which case all variables will be assumed to be continuous.
anCode	An integer vector containing the instruction list. The length of this vector is equal to *nCode*.
nCode	Number of items in the instruction list.
aiVars	An integer vector containing the variable index. The length of this vector is equal to *nVars*. This value may be set to '[]' if the variable index is consistent with the variable position in the variable array.
adVals	A double precision vector containing the value of each real number in the model. The length of this vector is equal to *nNums*.
adX0	A double precision vector containing starting values for each variable in the given model. The length of this vector is equal to *nVars*.
aiObj	An integer vector containing the beginning positions on the instruction list for each objective row. The length of this vector is equal to *nObjs*. Currently, there is only support for a single objective.
anObj	An integer vector containing the length of instruction code (i.e., the number of individual instruction items) for each objective row. The length of this vector is equal to *nObjs*. Currently, only a single objective function is allowed.
aiRows	An integer vector containing the beginning positions on the instruction list for each constraint row. The length of this vector is equal to *nCons*.
anRows	An integer vector containing the length of instruction code (i.e., the number of individual instruction items) for each constraint row. The length of this vector is equal to *nCons*.
adL	A double precision vector containing the lower bound of each variable.
adU	A double precision vector containing the upper bound of each variable.

LHS Arguments:

Name	**Description**
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSloadLPData()

Description:

Loads the given LP data into the *LSmodel* data structure.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSloadLPData', iModel, nCons, nVars,
nObjsense, dObjconst, adC, adB, achContypes, nAnnz, aiAcols, acAcols,
adCoef, aiArows, adL, adU)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the problem data.
nCons	Number of constraints in the model.
nVars	Number of variables in the model.
nObjsense	An integer indicator stating whether the objective is to be maximized or minimized.
dObjconst	A double precision value to be added to the objective value.
adC	A double precision vector containing the objective coefficients.
adB	A double vector containing the constraint right-hand side coefficients.
achContypes	A character vector containing the type of each constraint.
nAnnz	The number of nonzeros in the constraint matrix.
aiAcols	An integer vector containing the index of the first nonzero in each column.
acAcols	An integer vector containing the length of each column.
adACoef	A double precision vector containing the nonzero coefficients of the constraint matrix.
aiArows	An integer vector containing the row indices of the nonzeros in the constraint matrix.
adL	A double precision vector containing the lower bound of each variable.
AdU	A double precision vector containing the upper bound of each variable.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Remark:

- *LSXloadLPData(),* which admits the coefficient matrix in MATLAB's sparse form, can also be used as an alternative.

LSloadNameData()

Description:

Loads the given name data (e.g., row and column names), into the *LSmodel* data structure.

MATLAB Prototype:

>> [nStatus] = mxlindo('LSloadNameData', iModel, szTitle, szObjName, szRhsName, szRngName, szBndname, aszConNames, aszVarNames, aszConeNam

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the problem data.
szTitle	A string containing the title of the problem.
szObjName	A string containing the name of the objective.
szRhsName	A string containing the name of the right-hand side vector.
szRngName	A string containing the name of the range vector.
szBndname	A string containing the name of the bounds vector.
aszConNames	Reserved for future use. Currently, should be an empty vector.
aszVarNames	Reserved for future use. Currently, should be an empty vector.
aszConeNames	Reserved for future use. Currently, should be an empty vector.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSloadNLPData()

Description:

Loads a nonlinear program's data into the model data structure.

MATLAB Prototype:

```
>> "[nErrorCode] = mxLINDO('LSloadNLPData', iModel, aiCols, acCols,
adCoef, aiRows, nObjcnt, aiObjndx, adObjcoef)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the problem data.
aiCols	An integer vector containing the index of the first nonlinear nonzero in each column. This vector must have *nVars*+1 entries, where *nVars* is the number of variables. The last entry will be the index of the next appended column, assuming one was to be appended.
acCols	An integer vector containing the number of nonlinear elements in each column.
adCoef	A double precision vector containing initial values of the nonzero coefficients in the (Jacobian) matrix. It may be set to [ ], in which case, LINDO API will compute an initial matrix.
aiRows	An integer vector containing the row indices of the nonlinear elements.
nObjcnt	An integer containing the number of nonlinear variables in the objective.
aiObjndx	An integer vector containing the column indices of nonlinear variables in the objective function.
adObjCoef	A double precision vector containing the initial nonzero coefficients in the objective. It may be set to [ ], in which case, LINDO API will compute an initial gradient vector.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSloadQCData()

Description:

Loads quadratic program data into the *LSmodel* data structure.

Returns:

0 if successful, else one of the error codes listed in Appendix A, *Error Codes*.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSloadQCData', iModel, nQCnnz, aiQCrows,
aiQCvars1, aiQCvars2, adQCcoef)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the quadratic data.
nQCnnz	The total number of nonzeros in quadratic coefficient matrices.
aiQCrows	A vector containing the index of the constraint associated with each nonzero quadratic term. This vector must have *nQCnnz* entries.
aiQCvars1	A vector containing the index of the first variable defining each quadratic term. This vector must have *nQCnnz* entries.
aiQCvars2	A vector containing the index of the second variable defining each quadratic term. This vector must have *nQCnnz* entries.
adQCcoef	A vector containing the nonzero coefficients in the quadratic matrix. This vector must also have *nQCnnz* entries.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSloadSemiContData()

Description:

Loads semi-continuous data into the *Lsmodel* data structure.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSloadSemiContData', iModel, nSC, iVarndx,
ad1, adu)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the semi-continuous data.
nSC	The number of semi-continuous variables.
iVarndx	A vector containing the indices of semi-continuous variables. This vector must have *nSC* entries.
ad1	A vector containing the lower bound associated with each semi-continuous variable. This vector must also have *nSC* entries.
adu	A vector containing the upper bound associated with each semi-continuous variable. This vector must also have *nSC* entries.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSloadSETSData()

Description:

Loads special sets data into the *Lsmodel* data structure.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSloadSETSData', iModel, nSETS, szSETStype,
aiCARDnum, aiSETSbegcol, aiSETScols)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the set data.
nSETS	Number of sets to load.

szSETStype	A character vector containing the type of each set. Valid values for each set are
	`LS_MIP_SET_CARD`
	`LS_MIP_SET_SOS1`
	`LS_MIP_SET_SOS2`
	`LS_MIP_SET_SOS3`

aiCARDnum	An integer vector containing set cardinalities. This vector must have *nSETS* entries. The set cardinalities are taken into account only for sets with szSETStype[i] = LS_MIP_SET_CARD.
aiSETSbegcol	An integer vector containing the index of the first variable in each set. This vector must have nSETS+1 entries. The last entry will be the index of the next appended set, assuming one was to be appended. If aiSETSbegcol[i] < aiSETSbegcol [i-1], then LSERR_ERROR_IN_INPUT is returned.
aiSETScols	An integer vector containing the indices of variables in each set. If any index is not in the range [0, nVars -1], LSERR_INDEX_OUT_OF_RANGE is returned.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSloadVarType()

Description:

Loads the variable types data into the *LSmodel* data structure. This replaces the routine previously named *LSloadMIPData*().

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSloadVarType', iModel, achVartypes)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* in which to place the MIP data.
achVartypes	A character vector containing the type of each variable.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSloadStringData()

Description:

Loads a vector of strings into the *LSmodel* data structure.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSloadStringData', iModel, nStrings,
vStrings)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nStrings	An integer indicating the number of strings to be loaded.
vStrings	A vector containing the strings to be loaded.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSbuildStringData()

Description:

Finalizes the loading of the string data and build the string values.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSbuildStringData', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSdeleteStringData()

Description:

Deletes the string values data.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSdeleteStringData', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSloadString()

Description:

Loads a single string into the *LSmodel* data structure.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSloadString', iModel, szString)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szString	A variable containing the string to be loaded.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSdeleteString()

Description:

Deletes the complete string data, including the string vector and values.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSdeleteString', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetStringValue()

Description:

Retrieves a string value for a specified string index.

MATLAB Prototype:

```
>> [szValue, nStatus] = mxlindo('LSgetStringValue', iModel,
nStringIdx)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nStringIdx	An integer containing the index of the string whose value you wish to retrieve.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.
szValue	A string variable containing the string value.

Solver Initialization Routines

The following commands can be used to initialize the linear and mixed integer solvers.

LSloadBasis()

Description:

Provides a starting basis for the simplex method. A starting basis is frequently referred to as being a "warm start".

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSloadBasis', iModel, anCstatus, anRstatus)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* containing the model for which you are providing the basis.
anCstatus	An integer vector containing the status of each column in the given model.
anRstatus	An integer vector in which information about the status of the rows is to be placed.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSloadVarPriorities()

Description:

Provide priorities for each variable for use in branch-and-bound.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSloadVarPriorities', iModel, anCprior)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
anCprior	An integer vector containing the priority of each column in the given model.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSloadVarStartPoint()

Description:

Provide initial guesses for variable values.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSloadVarStartPoint', iModel, adPrimal)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
adPrimal	A double precision vector containing starting values for each variable in the given model.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSloadBlockStructure()

Description:

Provides a block structure for the constraint matrix by specifying block memberships of each variable and constraint.

MATLAB Prototype:

```
>> [nBlock, anRblock, anCblock, nType, nStatus] =
mxlindo('LSloadBlockStructure', iModel))
```

RHS Arguments:

Name	Description
iModel	A user assigned integer referring to an instance of *LSenv*.

LHS Arguments:

Name	Description
nBlock	An integer scalar that contains the number of blocks to decompose the model matrix into (Sensible only if nType = LS_LINK_BLOCKS_NONE).
anRblock	An integer vector in which information about the block membership of the constraints is to be placed. The i-th element of this array returns information on the i-th constraint as follows: 0: The row is a member of the linking (row) block. k>0: The row is a member of the k-th block. Where 1 <= k <= nBlock.
anCblock	An integer vector in which information about the block membership of the variables is to be placed. The j-th element of this array contains information on the j-th column as follows: 0: The column is a member of the linking (column) block. k>0: The column is a member of the k-th block. where 1 <= k <= nBlock.
nType	An integer returning the type of the decomposition.
nStatus	An integer error code. If successful, nStatus will be 0 on return. A list of possible error codes may be found in Appendix A.

LSreadVarPriorities()

Description:

Provide branching priorities for integer variables from a disk file.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSreadVarPriorities', iModel, szFname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szFname	The name of the file from which to read the variable priorities.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSreadVarStartPoint()

Description:

Provides initial values for variables from a file.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSreadVarStartPoint', iModel, szFname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szFname	The name of the file from which to read the initial values for the variables.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Optimization Routines

The following commands can be used to optimize a linear or mixed-integer program.

LSoptimize()

Description:

Optimizes a continuous model by a given method.

MATLAB Prototype:

```
>> [nSolStat, nStatus] = mxlindo('LSoptimize', iModel, nMethod)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* you wish to optimize.
nMethod	An integer to specify the type of solver to use. See the definition of *LSoptimize*() in Chapter 2, *Function Definitions*.

LHS Arguments:

Name	Description
nSolStat	An integer indicating the status of the solution.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSsolveGOP()

Description:

Optimizes a GOP model.

MATLAB Prototype:

```
>> [nSolStat, nStatus] = mxlindo('LSsolveGOP', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* you wish to optimize.

LHS Arguments:

Name	Description
nSolStat	An integer indicating the status of the GOP solution.
nStatus0	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSsolveMIP()

Description:

Optimizes a mixed-integer programming model using branch-and-bound.

MATLAB Prototype:

```
>> [nSolStat, nStatus] = mxlindo('LSsolveMIP', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* you wish to optimize.

LHS Arguments:

Name	Description
nSolStat	An integer indicating the status of the MIP solution.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Solution Query Routines

The following commands can be issued to retrieve information on the solution of the specified model:

Note: LINDO API uses the C-language type indexing of arrays. Therefore, any index set retrieved will start with *zero***.**

LSgetBasis()

Description:

Gets information about the basis that was found after optimizing the given model.

MATLAB Prototype:

```
>> [anCstatus, anRstatus, nStatus] = mxlindo('LSgetBasis', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* containing the model for which you are retrieving the basis.

LHS Arguments:

Name	Description
anCstatus	An integer vector containing the status of each column in the given model.
anRstatus	An integer vector in which information about the status of the rows is to be placed.

nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetDualSolution()

Description:

Returns the value of the dual variables for a given model.

MATLAB Prototype:

```
>> [adDual, nStatus] = mxlindo('LSgetDualSolution', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adDual	A double precision vector in which the dual solution is to be placed.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetInfo()

Description:

Returns model or solution information about the current state of the LINDO API solver after model optimization is completed. This function cannot be used to access callback information.

MATLAB Prototype:

```
>> [dValue, nStatus] = mxlindo('LSgetInfo', iModel, nQuery);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nQuery	The information desired from LINDO API. For possible values, see the definition of this function in Chapter 2, *Function Definitions*.

LHS Arguments:

Name	Description
dValue	A double precision scalar or a vector depending on the type of query.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetMIPBasis()

Description:

Gets information about the basis that was found after optimizing the LP relaxation of the node that yielded the optimal solution of a given MIP model.

MATLAB Prototype:

```
>> [anCstatus, anRstatus, nStatus] = mxlindo('LSgetMIPBasis', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* containing the model for which you are retrieving the basis.

LHS Arguments:

Name	Description
anCstatus	An integer vector containing the status of each column in the given model.
anRstatus	An integer vector in which information about the status of the rows is to be placed.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetMIPDualSolution()

Description:

Returns the value of the dual variables for a given MIP model.

MATLAB Prototype:

```
>> [adDual, nStatus] = mxlindo('LSgetMIPDualSolution', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adDual	A double precision vector in which the dual solution is to be placed.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetMIPPrimalSolution()

Description:

Gets the current solution for a MIP model.

MATLAB Prototype:

```
>>[ adPrimal, nStatus] = mxlindo('LSgetMIPPrimalSolution', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adPrimal	A double precision vector in which the primal solution is to be placed.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A, *Error Codes*.

LSgetMIPReducedCosts()

Description:

Gets the current reduced cost for a MIP model.

MATLAB Prototype:

```
>>[ adRedCost, nStatus] = mxlindo('LSgetMIPReducedCosts', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adRedCost	A double precision vector in which the reduced cost is to be placed.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetMIPSlacks()

Description:

Gets the slack values for a MIPmodel.

MATLAB Prototype:

```
>> [adSlacks, nStatus] = mxlindo('LSgetMIPSlacks', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adSlacks	A double precision vector in which the MIP slacks are to be placed.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetPrimalSolution()

Description:

Returns the value of the primal variables for a given model.

MATLAB Prototype:

```
>> [adPrimal, nStatus] = mxlindo('LSgetPrimalSolution', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adPrimal	A double precision vector in which the primal solution is to be placed.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Remarks:

- Error code *LSERR_INFO_NOT_AVAILABLE* -the requested info not available- is returned if any solution access routines are called after the optimization halts without computing a solution. The most common causes for not having a solution after the optimization are:
  Optimization halted due to a time or iteration limit,
  Optimization halted due to numerical errors,
  Optimization halted due to CTRL-C (user break),
  Presolver has determined the problem to be infeasible or unbounded.

In all these cases, the optimizer will return an associated error code (e.g., LSERR_ITER_LIMIT). During subsequent steps of user's application the type of the last error code returned by the optimizer can be accessed via LSgetInfo() function.

LSgetReducedCosts()

Description:

Returns the value of the reduced costs for a given model.

MATLAB Prototype:

```
>> [adRedcosts, nStatus] = mxlindo('LSgetReducedCosts', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adRedcosts	A double precision vector in which the reduced costs are to be placed.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetReducedCostsCone()

Description:

Returns the reduced cost of all cone variables of a given model.

MATLAB Prototype:

```
>> [adRedcosts, nStatus] = mxlindo('LSgetReducedCostsCone', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adRedcosts	A double precision vector in which the reduced costs of the variables are to be returned.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetSlacks()

Description:

Returns the value of the primal slacks for a given model.

MATLAB Prototype:

```
>> [adSlacks, nStatus] = mxlindo('LSgetSlacks', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adSlacks	A double precision vector in which the primal slacks are to be placed.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetSolution()

Description:

Gets the solution specified by the third argument.

MATLAB Prototype:

```
>> [adValues, nStatus] = mxlindo('LSgetSolution', iModel, nWhich);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nWhich	An integer parameter specifying the solution to be retrieved. Refer to Chapter 2 for possible values.

LHS Arguments:

Name	Description
adValues	A double precision vector in which the specified solution is to be placed.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Model Query Routines

The following commands can be issued to retrieve information on the specified model:

Note: LINDO API uses the C-language type indexing of arrays. Therefore, index set retrieved may contain *zero* as index value**.**

LSgetConeDatai()

Description:

Retrieve data for cone *i*.

MATLAB Prototype:

```
>> [achConeType, iNnz, iCols, nStatus] = mxlindo('LSgetConeDatai',
iModel, iCone);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
iCone	The index of the cone to retrieve the data for.

LHS Arguments:

Name	Description
achConeType	A character variable that returns the constraint's type. The returned value will be ''Q', or 'R'.
iNnz	An integer variable that returns the number of variables characterizing the cone.
iCols	An integer vector that returns the indices of variables characterizing the cone.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetConeIndex()

Description:

Gets the index of a cone with a specified name.

MATLAB Prototype:

```
>> [iCone, nStatus] = mxlindo('LSgetConeIndex', iModel, szConeName);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szConeName	A string containing the name of the cone for which the index is requested.

LHS Arguments:

Name	Description
iCone	An integer scalar that returns the index of the cone requested.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetConeNamei()

Description:

Gets the name of a cone with a specified index.

MATLAB Prototype:

```
>> [achConeName, nStatus] = mxlindo('LSgetConeNamei', iModel, iCone);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
iCone	Index of the cone whose name is to be retrieved.

LHS Arguments:

Name	Description
achConeName	A character array that contains the cone's name with a null terminator.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetConstraintDatai()

Description:

Gets data on a specified constraint.

MATLAB Prototype:

```
>> [chContype, chIsNlp, dB, nStatus] =
mxlindo('LSgetConstraintDatai', iModel , iCon);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
iCon	An integer containing the index of the constraint whose name you wish to retrieve.

LHS Arguments:

Name	Description
chContype	A character variable that returns the constraint's type. The returned value will be ''L', 'E', 'G', or 'N', for less-than-or-equal-to, equal to, greater-than-or-equal-to, or neutral, respectively.
chIsNlp	A character that returns 0 if the constraint is linear and 1 if it is nonlinear.
dB	A double precision variable that returns the constraint's right-hand side value.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetConstraintIndex()

Description:

Retrieves the internal index of a specified constraint name.

Prototype:

```
>> [iCon, nStatus] = mxlindo('LSgetConstraintIndex', iModel,
szConname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szConname	A character string containing the name of the constraint.

LHS Arguments:

Name	Description
iCon	An integer that returns the constraint's index.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetConstraintNamei()

Description:

Retrieves the name of a constraint, given its index number.

Prototype:

```
>> [szConname, nStatus] = mxlindo('LSgetConstraintNamej', iModel,
iCon)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
iCon	An integer containing the index of the constraint whose name you wish to retrieve.

LHS Arguments:

Name	Description
szConname	A character string that returns the constraint's name.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetLPConstraintDatai()

Description:

Retrieves the formulation data for a specified constraint in a linear or mixed integer linear program. Individual LSH entries may be set to '[]' if associated items are not required.

MATLAB Prototype:

```
>> [chContype, dB, nNnz, aiVar, adAcoef, nStatus] =
mxlindo('LSgetLPConstraintDatai', iModel , iCon);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
iCon	An integer containing the index of the constraint whose name you wish to retrieve.

LHS Arguments:

Name	Description
chContype	A character that returns the constraint's type. Values returned are 'L' for less-than-or-equal-to, 'E' for equal-to, 'G' for greater-than-or-equal-to, or 'N' for neutral.
dB	A double precision quantity that returns the constraint's right-hand side coefficient.
nNnz	An integer that returns the number of nonzero coefficients in the constraint.
aiVar	An integer vector that contains the indices of the variables to compute the partial derivatives for.
adAcoef	A vector containing nonzero coefficients of the new constraints.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetLPData()

Description:

Returns the formulation data for a given linear programming model.

MATLAB Prototype:

```
>> [nCons, nVars, nObjsense, dObjconst, adC, adB, achContypes,
aiAcols, acAcols, adCoef, aiArows, adL , adU, nStatus] =
mxlindo('LSgetLPData', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* from which to retrieve the problem data.

LHS Arguments:

Name	Description
nCons	Number of constraints in the model.
nVars	Number of variables in the model.
nObjsense	An indicator stating whether the objective is to be maximized or minimized.
dObjconst	A constant value to be added to the objective value.
adC	A double precision vector containing the objective coefficients.
adB	A double vector containing the constraint right-hand side coefficients.
achContypes	A character vector containing the type of each constraint.
aiAcols	An integer vector containing the index of the first nonzero in each column.
acAcols	An integer vector containing the length of each column.
adCoef	A double precision vector containing the nonzero coefficients of the constraint matrix.
aiArows	An integer vector containing the row indices of the nonzeros in the constraint matrix.
adL	A double precision vector containing the lower bound of each variable.
adU	A double precision vector containing the upper bound of each variable.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Remark:

- *LSXgetLPData()*, which retrieves the coefficient matrix in MATLAB's sparse form, can also be used as an alternative.

LSgetLPVariableDataj()

Description:

Retrieves the formulation data for a specified variable. Individual LHS entries may be set to '[]' if associated items are not required.

MATLAB Prototype:

```
>> [chVarType, dC, dL, dU, nAnnz, aiArows, nStatus] =
mxlindo('LSgetLPVariableDataj', iModel, iVar)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
iVar	An integer containing the index of the variable whose name you wish to retrieve.

LHS Arguments:

Name	Description
chVarType	A character that returns the variable's type. Values returned are 'B' for binary, 'C' for continuous, or 'I' for general integer.
dC	A double precision quantity that returns the variable's objective coefficient.
dL	A double precision quantity that returns the variable's lower bound.
dU	A double precision quantity that returns the variable's upper bound.
nAnnz	An integer that returns the number of nonzero constraint coefficients in the variable's column.
aiArows	An integer vector containing the row indices of the nonzeros in the new columns.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetNameData()

Description:

Returns the names—objective, right-hand side vector, range vector, bound vector, constraints, and variables—of a given model.

MATLAB Prototype:

```
>> [szTitle, szObjname, szRhsname, szRngname, szBndname, aszConnames,
achConNameData, aszVarnames, achVarNameData, nStatus] =
mxlindo('LSgetNameData', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
szTitle	A character array that returns the title of the problem. A model's title can be of any length, so be sure to allocate sufficient space to store the title you originally passed to LINDO API. The returned title will be null terminated.
szObjname	A character array that will return the name of the objective, null terminated.
szRhsname	A character array that returns the name of the right-hand side vector, null terminated.
szRngname	A character array that returns the name of the range vector, null terminated.
szBndname	A character array that returns the name of the bound vector, null terminated.
aszConnames	Reserved for future use. Currently, should be an empty vector.
achConNameData	Reserved for future use. Currently, should be an empty vector.
aszVarnames	Reserved for future use. Currently, should be an empty vector.
achVarNameData	Reserved for future use. Currently, should be an empty vector.

LSgetNLPConstraintDatai()

Description:

Gets data about the nonlinear structure of a specific row of the model.

MATLAB Prototype:

```
>> [nColcnt,aiColndx,adCoef,nErrorCode] = mxLINDO(
'LSgetNLPConstraintDatai', iModel, iCon)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* from which to retrieve the problem data.
iCon	An integer referring to the index of the constraint whose NLP data will be retrieved.

LHS Arguments:

Name	Description
nColcnt	An integer vector returning the number of nonlinear columns in the specified row.
aiColndx	An integer vector returning the column indices of the nonlinear nonzeros in the specified row.
adCoef	A double precision vector returning the current values of the nonzero coefficients of the specified row in the (Jacobian) matrix.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetNLPData()

Description:

Gets data about the nonlinear structure of a model, essentially the reverse of *LSloadNLPData*().

MATLAB Prototype:

```
>> [aiCols, acCols, adCoef, aiRows, nObj, aiObj, adObjCoef,
achConType, nStatus] = mxLINDO('LSgetNLPData',iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* from which to retrieve the problem data.

LHS Arguments:

Name	Description
aiCols	An integer vector returning the index of the first nonlinear nonzero in each column. The last entry will be the index of the next appended column, assuming one was to be appended.
acCols	An integer vector returning the number of nonlinear elements in each column.
adCoef	A double precision vector returning the current values of the nonzero coefficients in the (Jacobian) matrix.
aiRows	An integer vector returning the row indices of the nonlinear nonzeros in the coefficient matrix.
nObj	An integer returning the number of nonlinear variables in the objective function.
aiObj	An integer vector returning column indices of the nonlinear terms in the objective.
adObjCoef	A double precision vector returning the current partial derivatives of the objective corresponding to the variables *aiObj* [ ].
achConType	A character array whose elements indicate whether a constraint has nonlinear terms or not. If *achConType* [*i*] > 0, then constraint *i* has nonlinear terms.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetNLPObjectiveData()

Description:

Gets data about the nonlinear structure of the objective function of the model.

MATLAB Prototype:

```
>> [nObjcnt,aiColndx,adCoef,nErrorCode] = mxLINDO(
'LSgetNLPConstraintDatai', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* from which to retrieve the problem data.

LHS Arguments:

Name	Description
nObjcnt	An integer vector returning the number of nonlinear columns in the objective row.
aiColndx	An integer vector returning the column indices of the nonlinear nonzeros in the objective row.
adCoef	A double precision vector returning the current values of the nonzero coefficients of the gradient of the objective.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetNLPVariableDataj()

Description:

Gets data about the nonlinear structure of a specific column of the model.

MATLAB Prototype:

```
>> [nRowcnt,aiRowndx,adCoef,nErrorCode] = mxLINDO(
'LSgetNLPVariableDataj', iModel, iVar)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* from which to retrieve the problem data.
iVar	An integer referring to the index of the variable whose NLP data will be retrieved.

LHS Arguments:

Name	Description
nRowcnt	An integer vector returning the number of nonlinear rows in the specified variable's column.
aiRowndx	An integer vector returning the row indices of the nonlinear nonzeros in the specified variable's column.
adCoef	A double precision vector returning the current values of the nonzero coefficients of the specified column in the (Jacobian) matrix.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetQCData()

Description:

Retrieves quadratic program data in a given model.

MATLAB Prototype:

```
>> [nQCnnz, aiQCrows, aiQCvars1, aiQCvars2, adQCcoef, nStatus] =
mxlindo('LSgetQCData', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* from which to retrieve the problem data.

LHS Arguments:

Name	Description
nQCnnz	The total number of nonzeros in quadratic coefficient matrices.
aiQCrows	A vector containing the index of the constraint associated with each nonzero quadratic term.
aiQCvars1	A vector containing the index of the first variable defining each quadratic term.
aiQCvars2	A vector containing the index of the second variable defining each quadratic term. This vector will have *nQCnnz* entries.
adQCcoef	A vector containing the nonzero coefficients in the quadratic matrix. This vector will also have *nQCnnz* entries.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetQCDatai()

Description:

Retrieves quadratic program data of a single constraint in a given model.

MATLAB Prototype:

```
>> [nQCnnz, aiQCvars1, aiQCvars2, adQCcoef, nStatus] =
mxlindo('LSgetQCDatai', iModel, iCon)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* from which to retrieve the problem data.
iCon	An integer referring to the index of the constraint whose quadratic data will be retrieved.

LHS Arguments:

Name	Description
nQCnnz	The number of nonzeros in the quadratic coefficient matrix of the specified constraint.
aiQCvars1	A vector containing the index of the first variable defining the quadratic term. This vector will have *nQCnnz* entries.
aiQCvars2	A vector containing the index of the second variable defining the quadratic term. This vector will have *nQCnnz* entries.
adQCcoef	A vector containing the nonzero coefficients in the quadratic matrix. This vector will have *nQCnnz* entries.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetSemiContData()

Description:

Retrieves the semi-continuous data from an *LSmodel* data structure.

MATLAB Prototype:

```
>> [iNvars, iVarndx, ad1, adu, nStatus] =
mxlindo('LSgetSemiContData', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* from which to retrieve the problem data.

LHS Arguments:

Name	Description
iNvars	An integer variable to return the number of semi-continuous variables.
iVarndx	An integer vector to return the indices of semi-continuous variables.
ad1	A vector to return the lower bounds of semi-continuous variables.
adu	A vector to return the upper bounds of semi-continuous variables.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetSETSData()

Description:

Retrieves sets data from an *LSmodel* data structure.

MATLAB Prototype:

```
>> [iNsets, iNtnz, achSETtype, iCardnum, iNnz, iBegset, iVarndx,
nStatus] = mxlindo('LSgetSETSData', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* from which to retrieve the problem data.

LHS Arguments:

Name	Description
iNsets	An integer variable to return the number of sets in the model.
iNtnz	An integer variable to return the total number of variables in the sets.
achSETtype	A character array to return the type of sets in the model. The size of this array should be at least *(iNsets)*
iCardnum	An integer array to return the cardinalities of sets in the model. The size of this array should be at least *(iNsets)*
iNnz	An integer array to return the number of variables in each set in the model. The size of this array should be at least *(iNsets)*
iBegset	An integer array returning the index of the first variable in each set. This vector must have (iNsets + 1) entries, where *iNsets* is the number of sets in the model. The last entry will be the index of the next appended set, assuming one was to be appended.
iVarndx	An integer vector returning the indices of the variables in the sets. You must allocate at least one element in this vector for each <variable,set> tuple (i.e. at least iNtnz elements are required.)
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetSETSDatai()

Description:

Retrieves the data for set *i* from an *LSmodel* data structure.

MATLAB Prototype:

```
>> [achSETType, iCardnum, iNnz, iVarndx, nStatus] =
mxlindo('LSgetSETSDatai', iModel, iSet)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* from which to retrieve the problem data.
iSet	The index of the set to retrieve the data for.

LHS Arguments:

Name	Description
achSETType	A character variable to return the set type.
iCardnum	An integer variable to return the set cardinality.
iNnz	An integer variable to return the number of variables in the set.
iVarndx	An integer vector to return the indices of the variables in the set. This vector should have at least *(iNnz)* elements.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetVariableIndex()

Description:

Retrieves the internal index of a specified variable name.

Prototype:

```
>> [iVar, nStatus] = mxlindo('LSgetVariableIndex', iModel, szVarname)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szVarname	A character string containing the name of the variable.

LHS Arguments:

Name	Description
iVar	An integer that returns the variable's index.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetVariableNamej()

Description:

Retrieves the name of a variable, given its index number.

Prototype:

```
>> [szVarname, nStatus] = mxlindo('LSgetVariableNamej', iModel, iVar)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *Lsmodel*.
iVar	An integer containing the index of the variable whose name you wish to retrieve.

LHS Arguments:

Name	Description
szVarname	A character string that returns the variable's name.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetVarStartPoint()

Description:

Retrieves the values of the initial primal solution.

MATLAB Prototype:

```
>> [adPrimal, nStatus] = mxlindo('LSgetVarStartPoint', iModel);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adPrimal	A double precision vector that contains the primal solution at which the objective function will be evaluated.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetVarType()

Description:

Gets the variable type for a MIP model.

MATLAB Prototype:

```
>> [achVartypes,nCont,nBin,nGin,nStatus] = mxlindo('LSgetVarType',
iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* from which to retrieve the problem data.

LHS Arguments:

Name	Description
achVartypes	A character vector containing the type of each variable.
nCont	A scalar indicating the number of continuous variables in the model.
nBin	A scalar indicating the number of binary variables in the model.
nGin	A scalar indicating the number of general integer variables in the model.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Model Modification Routines

The following commands can be issued to modify an existing model *iModel* in several ways**.** Since the modification routines reset the solution status of the model to its default, the resident solution may not be optimal.

LSaddCones ()

Description:

Adds cones to a given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSaddCones', iModel, nCone, szConeTypes,
cConenames, aiConebegcol, aiConecols)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel* from which to retrieve the problem data.
nCone	An integer containing the number of cones to append.
szConeTypes	A character array containing the type of each cone to be added to the model.
cConenames	Reserved for future use. Currently, should be empty vector.
aiConebegcol	An integer vector containing the index of the first variable in each new cone. This vector must have *nCone* +1 entries. The last entry should be equal to the number of variables in the added cones.
aiConecols	An integer vector containing the indices of the variables in the new cones.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSaddConstraints()

Description:

Adds constraints to a given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSaddConstraints', iModel, nCons,
achContypes, aszConnames, aiArows, adAcoef, aiAcols, adB)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nCons	An integer containing the number of constraints to append.
achContypes	A character vector containing the type of each constraint to be added to the model.
aszConnames	A vector of null terminated strings containing the name of each new constraint.
aiArows	An integer vector containing the index of the first nonzero element in each new constraint.
adAcoef	A vector containing nonzero coefficients of the new constraints.
aiAcols	An integer vector containing the column indices of the nonzeros in the new constraints.
adB	A double precision vector containing the right-hand side coefficients for each new constraint.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSaddSETS()

Description:

Adds sets to a given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSaddSETS', iModel, nSETS, szSETStypes,
aiCARDnum, aiSETSbegcol, aiSETScols)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nSETS	An integer containing the number of sets to add.
szSETStypes	A character array containing the type of each set to be added to the model.
aiCARDnum	An integer array containing the cardinalities of the sets to be added.
aiSETSbegcol	An integer vector containing the index of the first variable in each new set. This vector must have *nSETS* +1 entries. The last entry should be equal to the total number of variables in the new sets.
aiSETScols	An integer vector containing the indices of the variables in the new sets.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSaddVariables()

Description:

Adds variables to a given model. If both constraints and variables need to be added to a model and adding the new information in column format is preferred, then this routine can be called after first calling *LSaddConstraints*().

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSaddVariables', iModel, nVars, achVartypes,
aszVarnames, aiAcols, acAcols, adAcoef, aiArows, adC, adL, adU)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nVars	The number of variables to append to the model.
achVartypes	A character vector indicating the types of each variable to be added to the model.
aszVarnames	A vector of null terminated strings containing the name of each new variable.
aiAcols	An integer vector containing the index of the first nonzero element in each new column.
acAcols	An integer vector containing the length of each column.
adAcoef	A double precision vector containing the nonzero coefficients of the new columns.
aiArows	An integer vector containing the row indices of the nonzeros in the new columns.
adC	A double precision vector containing the objective coefficients for each new variable.
adL	A double precision vector containing the lower bound of each new variable.
adU	A double precision vector containing the upper bound of each new variable.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSaddQCterms()

Description:

Adds quadratic elements to the given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSaddQCterms', iModel, nQCnonzeros,
vaiQCconndx, vaiQCvarndx1, vaiQCvarndx2, vadQCcoef)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nQCnonzeros	The total number of nonzeros in quadratic coefficient matrices to be added.
vaiQCconndx	A vector containing the index of the constraint associated with each nonzero quadratic term. This vector must have *nQCnonzeros* entries.
vaiQCvarndx1	A vector containing the indices of the first variable defining each quadratic term. This vector must have *nQCnonzeros* entries.
vaiQCvarndx2	A vector containing the indices of the second variable defining each quadratic term. This vector must have *nQCnonzeros* entries.
vadQCcoef	A vector containing the nonzero coefficients in the quadratic matrix. This vector must also have *nQCnonzeros* entries.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSaddNLPAj()

Description:

Adds NLP elements to the specified column for the given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSaddNLPAj', iModel, iVar1, nRows, vaiRows,
vadAj)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
iVar1	The index of the variable to which NLP elements will be added.
nRows	The total number of constraints for which NLP elements will be added.
vaiRows	An integer vector containing the row indices of the nonlinear elements. The indices are required to be in ascending order.
vadAj	A double vector containing the initial nonzero coefficients of the NLP elements. If *vadAj* is NULL, the solver will set the initial values.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSaddNLPobj()

Description:

Adds NLP elements to the objective function for the given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSaddNLPobj', iModel, nCols, vaiCols,
vadColj)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nCols	The total number of variables for which NLP elements will be added.
vaiCols	A integer vector containing the variable indices of the nonlinear elements.
vadColj	A double vector containing the initial the initial nonzero coefficients of the NLP elements. If v*adColj* is NULL, the solver will set the initial values.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSdeleteCones()

Description:

Deletes a set of cones in the given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSdeleteCones', iModel, nCones, aiCones)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nCones	The number of cones in the model to delete.
aiCones	A vector containing the indices of the cones that are to be deleted.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSdeleteConstraints()

Description:

Deletes a set of constraints in the given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSdeleteConstraints', iModel, nCons, aiCons)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nCons	The number of constraints in the model to delete.
aiCons	A vector containing the indices of the constraints that are to be deleted.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSdeleteQCterms()

Description:

Deletes the quadratic terms in a set of constraints in the given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSdeleteQCterms', iModel, nCons, aiCons)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nCons	The number of constraints in the model for which the quadratic terms will be deleted.
aiCons	A vector containing the indices of the constraints whose quadratic terms are to be deleted.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSdeleteSemiContVars()

Description:

Deletes a set of semi-continuous variables in the given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSdeleteSemiContVars', iModel, nSC, SCndx)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nSC	The number of semi-continuous variables in the model to delete.
SCndx	A vector containing the indices of the semi-continuous variables that are to be deleted.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSdeleteSETS()

Description:

Deletes the sets in the given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSdeleteSETS', iModel, nSETS, SETSndx)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nSETS	The number of sets in the model to delete.
SETSndx	A vector containing the indices of the sets that are to be deleted.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSdeleteVariables()

Description:

Deletes a set of variables in the given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSdeleteVariables', iModel, nVars, aiVars)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nVars	The number of variables in the model to delete.
aiVars	A vector containing the indices of the variables that are to be deleted.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSdeleteAj()

Description:

Deletes all the elements in the specified column for the given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSdeleteAj', iModel, iVar1, nRows, vaiRows)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
iVar1	The index of the variable whose lements will be deleted.
nRows	The number of constraints at which elements will be deleted.
vaiRows	An integer vector containing the row indices of the elements to be deleted. The indices are required to be in ascending order.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSdeleteNLPobj()

Description:

Deletes NLP elements from the objective function for the given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSdeleteNLPobj', iModel, nCols, vaiCols)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nCols	The number of variables for which NLP elements will be deleted.
vaiCols	A vector containing the indices of the variables whose NLP elements are to be deleted.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSmodifyAj()

Description:

Modifies the coefficients for a given column at specified constraints.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSmodifyAj', iModel, iVar1, nCons, aiCons,
adAj)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
iVar1	The index of the variable to modify the constraint coefficients.
nCons	Number of constraints to modify.
aiCons	An array of the indices of the constraints to modify.
adAj	A double precision array containing the values of the new coefficients.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSmodifyCone()

Description:

Modifies the data for the specified cone.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSmodifyCone', iModel, cConeType, iConeNum,
iConeNnz, aiConeCols)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
cConeType	A character variable specifying the new type of the cone.
iConeNum	An integer scalar that refers to the index of the cone to modify.
iConeNnz	An integer scalar that refers to the number of variables characterizing the cone.
aiConeCols	An integer vector that keeps the indices of the variables characterizing the cone. Its size should be *iConeNnz*.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSmodifyConstraintType()

Description:

Modifies the senses of the selected constraints of a given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSmodifyConstraintType', iModel, nCons,
aiCons, achContypes)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nCons	The number of constraint senses to modify.
aiCons	An integer vector containing the indices of the constraints whose senses are to be modified.
achContypes	A character vector in which each element is either: ‘L’, ‘E’, ‘G’, or ‘N’ indicating each constraint's type.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSmodifyLowerBounds()

Description:

Modifies selected lower bounds in a given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSmodifyLowerBounds', iModel, nVars, aiVars,
adL)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nVars	The number of bounds in the model to modify.
aiVars	An integer vector containing the indices of the variables for which to modify the lower bounds.
adL	A double precision vector containing the new values of the lower bounds on the variables.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSmodifyObjConstant()

Description:

Modifies the objective's constant term for a specified model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSmodifyObjConstant', iModel , dObjconst);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
dObjconst	The new objective constant term.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSmodifyObjective()

Description:

Modifies selected objective coefficients of a given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSmodifyObjective', iModel, nVars, aiVars, adC)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nVars	Number of objective coefficients to modify.
aiVars	An integer vector containing a list of the indices of the objective coefficients to modify.
adC	A double precision vector containing the new values for the modified objective coefficients.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSmodifyRHS()

Description:

Modifies selected constraint right-hand sides of a given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSmodifyRHS', iModel, nCons, aiCons, adB)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nCons	The number of constraint right-hand sides to modify.
aiCons	An integer vector containing the indices of the constraints whose right-hand sides are to be modified.
adB	A double precision vector containing the new right-hand side values for the modified right-hand sides.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSmodifySemiContVars()

Description:

Modifies data of a set of semi-continuous variables in the given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSmodifySemiContVars', iModel, nSC, iVarndx,
ad1, adu)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nSC	The number of semi-continuous variables to modify.
iVarndx	An integer vector containing the indices of the variables whose data are to be modified.
ad1	A double precision vector containing the new lower bound values for the semi-continuous variables.
adu	A double precision vector containing the new upper bound values for the semi-continuous variables.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSmodifySET()

Description:

Modifies set data in the given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSmodifySET', iModel, cSETtype, iSETnum,
iSETnnz, aiSETcols)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
cSETtype	A character variable containing the new type for the specified set.
iSETnum	An integer variable containing the index of the set to apply the modification.
iSETnnz	An integer variable containing the number of variables in the set specified with *iSETnum*.
aiSETcols	An integer array containing the indices of variables in the set specified with *iSETnum.*

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSmodifyUpperBounds()

Description:

Modifies selected upper bounds in a given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSmodifyUpperBounds', iModel, nVars, aiVars, adU)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nVars	The number of bounds in the model to modify.
aiVars	A vector containing the indices of the variables for which to modify the upper bounds.
adU	A double precision vector containing the new values of the upper bounds on the variables.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSmodifyVariableType()

Description:

Modifies the types of the selected variables of a given model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSmodifyVariableType', iModel, nVars, aiVars,
achVartypes)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nVars	The number of variable types to modify.
aiVars	An integer vector containing the indices of the variables whose types are to be modified.
achVartypes	A character vector containing strings of length *nVars* specifying the types of the specified variables.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Model and Solution Analysis Routines

The routines in the section below allow you to analyze models and their solutions. For a more detailed overview, see the Chapter 10, *Analyzing Models and Solutions*.

LSfindBlockStructure ()

Description:

Examines the nonzero structure of the constraint matrix and tries to identify block structures in the model..

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSfindBlockStructure', iModel, nBlock, nType)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nBlock	An integer scalar that contains the number of blocks to decompose the model matrix into. (Sensible only if *nType* <> *LS_LINK_BLOCKS_NONE*.)
nType	An integer scalar indicating the type of decomposition requested. The possible values are identified with the following macros: *LS_LINK_BLOCKS_NONE*: Try total decomposition (no linking rows or columns). *LS_LINK_BLOCKS_COLS*: The decomposed model will have dual angular structure (linking columns). *LS_LINK_BLOCKS_ROWS*: The decomposed model will have block angular structure (linking rows). *LS_LINK_BLOCKS_BOTH*: The decomposed model will have both dual and block angular structure (linking rows and columns). *LS_LINK_BLOCKS_FREE*: Solver decides which type of decomposition to use.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A

LSfindIIS()

Description:

Determines an irreducibly inconsistent set (IIS) of constraints for an infeasible linear program. Any of the RHS arguments can be set to empty vectors if the corresponding information is not required.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSfindIIS', iModel, nLevel);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nLevel	An integer indicating the level of detail of the analysis in finding the IIS. Possible values are: *LS_NECESSARY_ROWS* = 1, *LS_NECESSARY_COLS* = 2, *LS_SUFFICIENT_ROWS* = 4, *LS_SUFFICIENT_COLS* = 8.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSfindIUS()

Description:

Determines an irreducibly unbounded set (IUS) of columns for an unbounded linear program.

MATLAB Prototype:

```
>> [nStatus) = mxlindo('LSfindIUS', iModel, nLevel);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nLevel	An integer indicating the level of detail of the analysis in finding the IUS. Possible values are: *LS_NECESSARY_COLS* = 2, *LS_SUFFICIENT_COLS* = 8.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetBestBounds ()

Description:

Finds the best-implied variable bounds for the specified model by improving the original bounds using extensive preprocessing and probing.

MATLAB Prototype:

```
>> [adBestL, adBestU, nStatus] = mxlindo('LSgetBestBounds', iModel);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adBestL	A double precision vector containing the best-implied lower bounds.
adBestU	A double precision vector containing the best implied upper bounds.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetBlockStructure ()

Description:

Retrieves the block structure, identified by *LSfindBlockStructure*(), in the model..

MATLAB Prototype:

```
>> [nBlock, anRblock, anCblock, nType, nStatus] =
mxlindo('LSgetBlockStructure', iModel))
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
nBlock	An integer scalar that contains the number of blocks to decompose the model matrix into (Sensible only if *nType* = *LS_LINK_BLOCKS_NONE*).
anRblock	An integer vector in which information about the block membership of the constraints is to be placed. The *i*-th element of this array returns information on the *i*-th constraint as follows: 0: The row is a member of the linking (row) block. *k*>0: The row is a member of the *k*-th block. where 1 <= *k* <= *nBlock*.
anCblock	An integer vector in which information about the block membership of the variables is to be placed. The *j*-th element of this array contains information on the *j*-th column as follows: 0: The column is a member of the linking (column) block. *k*>0: The column is a member of the *k*-th block. where 1 <= *k* <= *nBlock*.
nType	An integer returning the type of the decomposition.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Remarks:

- For a brief overview on decomposition and linking structures, refer to Chapter 10, *Analyzing Models and Solutions*.

LSgetBoundRanges ()

Description:

Retrieves the maximum allowable decrease and increase in the primal variables for which the optimal basis remains unchanged.

MATLAB Prototype:

```
>> [adDec, adInc, nStatus] = mxlindo('LSgetBoundRanges', iModel);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adDec	A double precision vector that keeps the maximum allowable decrease in the lower and upper bounds.
adInc	A double precision vector that keeps the maximum allowable increase in the lower and upper bounds.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetConstraintRanges ()

Description:

Retrieves the maximum allowable decrease and increase in the right-hand side values of constraints for which the optimal basis remains unchanged.

MATLAB Prototype:

```
>> [adDec, adInc, nStatus] = mxlindo('LSgetConstraintRanges', iModel);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adDec	A vector that keeps the maximum allowable decrease in the right-hand sides of constraints.
adInc	A vector that keeps the maximum allowable increase in the right-hand sides of constraints.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetIIS()

Description:

Retrieves the irreducibly inconsistent set (IIS) determined by *LSfindIIS*(). Any of the RHS arguments can be set to empty vectors if the corresponding information is not required.

MATLAB Prototype:

```
>> [nSuf_r, nIIS_r, aiCons, nSuf_c, nIIS_c, aiVars, anBnds, nStatus]
= mxlindo('LSgetIIS', iModel);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
nSuf_r	The number of constraints in the *sufficient* set.
nIIS_r	The number of rows in the IIS.
aiCons	A vector of size *nIIS_r* containing the indices of the rows in the IIS. The locations *aiCons*[1] to *aiCons*[*nSuf_r*] keep the indices of the sufficient rows.
nSuf_c	The number of column bounds in the *sufficient* set.
nIIS_c	The number of column bounds in the IIS.
aiVars	A vector of size *nIIS_c* containing the indices of the column bounds in the IIS. The locations *aiVars*[1] to *aiVars*[*nSuf_c*] store the indices of the members of the sufficient column bounds. Passing an empty matrix forces the algorithm to ignore the column bounds as the source of infeasibility.
anBnds	A vector of size *nIIS_c* indicating whether the lower or the upper bound of the variable is in the IIS. Its elements are *–1* for *lower* bounds and *+1* for *upper* bounds.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetIUS()

Description:

Retrieves the irreducibly unbounded set (IUS) of columns determined by a call to *LSfindIUS*().

MATLAB Prototype:

```
>> [nSuf, nIUS, aiVars, nStatus) = mxlindo('LSgetIUS', iModel);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
nSuf	The number of columns in the *sufficient* set.
nIUS	The number of columns in the IUS.
aiVars	A vector of size *nIUS* containing the indices of the columns in the IUS. The locations *aiVars*[1] to *aiVars*[*nSuf*] store the indices of the members of the sufficient set.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetObjectiveRanges ()

Description:

Retrieves the maximum allowable decrease and increase in objective function coefficients for which the optimal basis remains unchanged.

MATLAB Prototype:

```
>> [adDec, adInc, nStatus] = mxlindo('LSgetObjectiveRanges', iModel);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
adDec	A vector that keeps the maximum allowable decrease in the objective function coefficients.
adInc	A vector that keeps the maximum allowable increase in the objective function coefficients.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Advanced Routines

The routines in this section perform specialized functions for users who are developing customized solution procedures.

LSdoBTRAN()

Description:

Does a so-called backward transformation. That is, the function solves the linear system $B^T X = Y$, where B^T is the transpose of the current basis of the given linear program and Y is a user specified vector.

MATLAB Prototype:

```
>> [cXnz, aiX, adX, nStatus] = mxlindo('LSdoBTRAN', iModel, cYnz,
aiY, adY)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
cYnz	An integer containing the number of nonzeros in the right-hand side vector *Y*.
aiY	An integer vector containing the positions of the nonzeros in *Y*.
adY	A double precision vector containing the coefficients of the nonzeros in *Y*.

LHS Arguments:

Name	Description
cXnz	An integer containing the number of nonzeros in the solution vector *X*.
aiX	An integer vector containing the positions of the nonzeros in *X*.
adX	A double precision vector containing the coefficients of the nonzeros in *X*.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSdoFTRAN()

Description:

Does a so-called forward transformation. That is, the function solves the linear system $BX = Y$, where B is the current basis of the given linear program, and Y is a user specified vector.

MATLAB Prototype:

```
>> [cXnz, aiX, adX, nStatus] = mxlindo('LSdoFTRAN', iModel, cYnz,
aiY, adY)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
cYnz	An integer containing the number of nonzeros in the right-hand side vector *Y*.
aiY	An integer vector containing the positions of the nonzeros in *Y*.
adY	A double precision vector containing the coefficients of the nonzeros in *Y*.

LHS Arguments:

Name	Description
cXnz	An integer containing the number of nonzeros in the solution vector *X*.
aiX	An integer vector containing the positions of the nonzeros in *X*.
adX	A double precision vector containing the coefficients of the nonzeros in *X*.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LScalcConFunc()

Description:

Calculates the constraint activity at a primal solution .

MATLAB Prototype:

```
>> [dValue, nStatus] = mxlindo('LScalcConFunc', iModel, iCon,
adPrimal);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
iCon	An integer containing the index of the constraint whose activity is requested.
adPrimal	A double precision vector that contains the primal solution at which the constraint activity will be computed.

LHS Arguments:

Name	Description
dValue	A double precision variable that returns the constraint activity at the given primal solution.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LScalcObjFunc()

Description:

Calculates the objective function value at a primal solution .

MATLAB Prototype:

```
>> [dPobjval, nStatus] = mxlindo('LScalcObjFunc', iModel,adPrimal);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
adPrimal	A double precision vector that contains the primal solution at which the objective function will be evaluated.

LHS Arguments:

Name	Description
dPobjval	A double precision variable that returns the objective value for the given primal solution.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LScalcConGrad()

Description:

Calculates the partial derivatives of the function representing a constraint with respect to a set of primal variables.

MATLAB Prototype:

```
>> [adVar, nStatus] = mxlindo('LScalcConGrad', iModel, iCon,
adPrimal, nVar, aiVar);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
iCon	An integer containing the index of the constraint whose partial derivatives is requested.
adPrimal	A double precision vector that contains the primal solution at which the partial derivatives of the constraint will be evaluated.
nVar	An integer scalar indicating the number of variables to compute the partial derivatives.
aiVar	An integer vector that contains the indices of the variables to compute the partial derivatives for.

LHS Arguments:

Name	Description
adVar	A double precision vector that returns the partial derivatives of the variables indicated by *aiVar*[].
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LScalcObjGrad()

Description:

Calculates the partial derivatives of the objective function with respect to a set of primal variables.

MATLAB Prototype:

```
>> [adVar, nStatus] = mxlindo('LScalcObjGrad', iModel, adPrimal,
nVar, aiVar);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
adPrimal	A double precision vector that contains the primal solution at which the partial derivatives of the objective function will be evaluated.
nVar	An integer scalar indicating the number of variables to compute the partial derivatives.
aiVar	An integer vector that contains the indices of the variables to compute the partial derivatives for.

LHS Arguments:

Name	Description
adVar	A double precision vector that returns the partial derivatives of the variables indicated by aiVar[].
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Callback Management Routines

The routines in this section allow the user to set callback *m-functions* and manage callback information.

LSgetCallbackInfo()

Description:

Returns information about the current state of the LINDO API solver during model optimization. This routine is to be called from your user supplied callback *m-function* that was set with *LSsetCallback*().

MATLAB Prototype:

```
>> [dValue, nStatus] = mxlindo('LSgetCallbackInfo', iModel,
nLocation, nQuery);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*. This should be the same instance as was passed to your user callback function from LINDO API.
nLocation	The solver's current location. This parameter is passed to your callback function by LINDO API.
nQuery	The information desired from LINDO API. For possible values, see the definition of this function in Chapter 2, *Function Definitions*.

LHS Arguments:

Name	Description
dValue	A double precision scalar or a vector depending on the type of query.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSgetMIPCallbackInfo()

Description:

Returns information about the MIP solver. This routine is to be called from your user supplied callback functions that were established with calls *LSsetCallback*() and *LSsetMIPCallback*().

MATLAB Prototype:

```
>> [dValue, nStatus] = mxlindo('LSgetMIPCallbackInfo', iModel,
nQuery);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*. This should be the same instance as was passed to your user callback function from the LINDO API solver.
nQuery	The information requested from LINDO API. See the function definition in Chapter 2, *Function Definitions*, for the information available through this routine.

LHS Arguments:

Name	Description
dValue	A double precision scalar or a vector depending on the type of query.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSsetCallback()

Description:

Supplies LINDO API with the name of a user-supplied *m-function* that will be called at various points during the solution process. The user-supplied *m-function* can be used to report the progress of the solver routines to a user interface, interrupt the solver, etc.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSsetCallback', iModel, szCbfunc, szData);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szCbfunc	A character string referring to the name of the user supplied callback *m-function*.
szData	A dummy character string. Reserved for future use.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Remarks:

- The m-function *szCbfunc.m* should have the following MATLAB calling sequence:

  function *retval* = *szCbfunc*(*iModel*, *loc*, *szData*)

- The user need not be concerned about the types and values of the RHS arguments. mxLINDO will ensure that correct types and values are passed.
- The value returned by the callback function, *retval*, specifies if the solver should be interrupted or not. A return value different than zero will interrupt the solver.
- See *LMreadF.m* and the sample callback function *LMcback.m* that came with your mxLINDO distribution.

LSsetFuncalc()

Description:

Supplies LINDO API with a) the user-supplied M-function *szFuncalc* (see Chapter 7) that will be called each time LINDO API needs to compute a row value, and b) reference to the user data area to be passed through to the *szFuncalc* function.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSsetFuncalc', iModel , szFuncalc , iUserData
);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szFuncalc	The name of the M-function that computes the value of a specified nonlinear row. See the definition of *pFuncalc*() in Chapter 7, *Solving Nonlinear Programs*, for details of this function's prototype in C calling conventions.
iUserData	A reference to a "pass through" data area in which your calling application may place information about the functions to be calculated.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSsetGradcalc()

Description:

Supplies LINDO API with a) the user-supplied M-function *szGradcalc* (see Chapter 7, *Solving Nonlinear Programs*) that will be called each time LINDO API needs a gradient (i.e., vector of partial derivatives), and b) the reference to data area to be passed through to the gradient computing routine. This data area may be the same one supplied to *LSsetFuncalc*().

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSsetGradcalc', iModel , szGradcalc,
iUserData, nLenUseGrad, aiUseGrad);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szGradcalc	A string containing the name of the M-function that computes the gradients for specified nonlinear rows. See the definition of *pGradcalc*() in Chapter 7, *Solving Nonlinear Programs*, for details on this function's interface in C calling conventions .
iUserData	A reference to a "pass through" data area in which your calling application may place information about the functions to be calculated.
nLenUseGrad	An integer indicating how many nonlinear rows will make use of the *szGradcalc* function. 0 is interpreted as meaning that no functions use s*zGradcalc* function, thus meaning that partials on all functions are computed with finite differences. A value of -1 is interpreted as meaning the partials on all nonlinear rows will be computed through the s*zGradcalc* function. A value greater than 0 and less-than-or-equal-to the number of nonlinear rows is interpreted as being the number of nonlinear rows that make use of the s*zGradcalc* function. And, the list of indices of the rows that do so is contained in the following array, *aiUseGrad*.
aiUseGrad	An integer array containing the list of nonlinear rows that make use of the s*zGradcalc* function. You should set this value to '[]' if *nLenUseGrad* is 0 or -1. Otherwise, it should be an array of dimension *nLenUseGrad*, where *aiUseGrad*[*j*] is the index of the *j*-th row whose partial derivatives are supplied through the s*zGradcalc* function. A value of -1 indicates the objective row.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSsetMIPCallback()

Description:

Supplies LINDO API with the address of the callback *m-function* that will be called each time a new integer solution has been found to a mixed-integer model.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSsetMIPCallback', iModel, szMIPCbfunc,
szData);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szMIPCbfunc	A character string referring to the name of the user supplied callback *m-function*.
szData	A dummy character string. Reserved for future use.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Remarks:

- The m-function *szMIPCbfunc.m* should have the following calling sequence:

  function *retval* = *szMIPCbfunc*(*iModel*, *szData*, *pdObjval*, *adPrimal*)

- The MIP callback functions cannot be used to interrupt the solver, instead the general callback function set by *LSsetCallback*() routine should be used.
- See *LMreadF.m* and the sample callback function *LMcbMLP.m* that came with your mxLINDO distribution.

LSsetModelLogFunc()

Description:

Supplies the specified model with a) the user-supplied M-function *szLogfunc* that will be called each time LINDO API logs a message and b) the reference to the user data area to be passed through to the *szLogfunc* function.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSsetModelLogFunc', iModel, szLogfunc,
iUserData);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
szLogfunc	A string containing the name of the M-function that will be called to log messages.
iUserData	A reference to a "pass through" data area in which your calling application may place information about the functions to be calculated.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSsetUsercalc()

Description:

Supplies LINDO API with the addresses of a) the *pUsercalc*() that will be called each time LINDO API needs to compute the value of the user-defined function and b) the address of the user data area to be passed through to the *pUsercalc*() routine.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSsetUsercalc', iModel, iUsercalc,
iUserData);
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
iUsercalc	The subroutine that computes the value of a user-defined function.
iUserData	A "pass through" data area in which your calling application may place information about the functions to be calculated.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Auxiliary Routines

Two auxiliary routines have been added to the MATLAB interface in order to allow the users to load or retrieve linear and mixed integer models without being concerned about the sparse representation of the coefficient matrix as required by LINDO API. These routines are not part of LINDO API.

LSXgetLPData()

Description:

This routine is for accessing the data of model *iModel*. Its difference from "LSgetLPData" is that, it does not return the additional vectors *aiAcols*, ac*Acols*, and ai*Arows* used for sparse representation of the coefficient matrix. On return, the coefficient matrix is already in MATLAB's sparse form. The calling sequence is:

MATLAB Prototype:

```
>> [nObjsense, dObjconst, adC, adB, achContypes, adA, adL, adU,
nStatus] = mxlindo('LSXgetLPData', iModel)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.

LHS Arguments:

Name	Description
nObjsense	An indicator stating whether the objective is to be maximized or minimized.
dObjconst	A constant value to be added to the objective value.
adC	A double precision vector containing the objective coefficients.
adB	A double precision vector containing the RHS coefficients.
achContypes	A character vector containing the type of constraints.
adA	A matrix in MATLAB's sparse format representing the LP coefficient matrix.
adL	A double precision vector containing the lower bounds.
adU	A double precision vector containing the upper bounds.
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

LSXloadLPData()

Description:

The routine "LSXloadLPData" loads the data of a linear model *iModel* into LINDO API. It is different from "LSloadLPData" in the sense that the additional vectors *aiAcols*, ac*Acols*, and ai*Arows* are not required as input for sparse representation of the coefficient matrix. This routine already admits the coefficient matrix in MATLAB's sparse form.

MATLAB Prototype:

```
>> [nStatus] = mxlindo('LSXloadLPData', iModel, nObjsense, dObjconst,
adC, adB, achContypes, adA, adL, adU)
```

RHS Arguments:

Name	Description
iModel	An integer referring to an instance of *LSmodel*.
nObjsense	An indicator stating whether the objective is to be maximized or minimized.
dObjconst	A constant value to be added to the objective value.
adC	A double precision vector containing the objective coefficients.
adB	A double precision vector containing the RHS coefficients.
achContypes	A character vector containing the type of constraints.
adA	A matrix in MATLAB's sparse format representing the LP coefficient matrix.
adL	A double precision vector containing the lower bounds.
adU	A double precision vector containing the upper bounds.

LHS Arguments:

Name	Description
nStatus	An integer error code. If successful, *nStatus* will be 0 on return. A list of possible error codes may be found in Appendix A.

Sample MATLAB Functions

M-functions using mxLINDO

The LINDO API distribution package contains a number of sample *m-functions* that demonstrate how *mxLINDO* can be used in MATLAB to set up, solve, and query linear and nonlinear mixed-integer models with LINDO API. At the beginning of the chapter we gave an example of solving a linear program using the LMsolveM.m m file. We continue with some additional illustrations of using mxLINDO based m files.

Solving Quadratic Programs with LMsolveM.m

LMsolvem has an extended argument list for solving quadratically constrained quadratic programs (QCP) and retrieving their solutions using mxLINDO. Suppose, the data objects illustrated in Figure 10.3 have been constructed.

```
» A = [0         0         0         0
       1         1         1         1 ];
» b = [0.2000    1.0000]';
» c = [0.3000    0.2000   -0.4000    0.2000]';
» csense = 'LE';
» vtype  = 'CCCC';
» l=[]; u=[];
» QCrows = [0       0       0       0       0       0       0   ];
» QCvars1 = [0       0       0       1       1       2       3   ];
» QCvars2 = [0       1       2       1       2       2       3   ];
» QCCoef = [1.00    0.64    0.27    1.00    0.13    1.00    1.00];
```

Figure 10.3

These objects represent a QCP instance of the form:

$$\begin{aligned} \text{Minimize} \quad & c^T x + \tfrac{1}{2} x^T Q^0 x \\ \text{S.t.} \quad & a_i x + \tfrac{1}{2} x^T Q^i x \geq b \quad \text{for } i=1,..,m \\ & u \geq x \geq l \end{aligned}$$

To solve this QCP, issue the following command at the MATLAB prompt:

```
>> [x, y, s, dj, obj, solstat] = LMsolvem(A, b, c, csense, l, u,
vtype, QCrows, QCvars1, QCvars2, QCcoef)
```

As illustrated in Figure 10.4, the function returns the primal and dual solutions (*x,s)* and (*y,dj)*, the optimal objective value *obj*, and the optimization status flag *solstat*.

```
» [x, y, s, dj, pobj, solstat] = LMsolvem(A, b, adC, csense, l, u,
vtype, QCrows, QCvars1, QCvars2, QCcoef)

x =

    0.0000
    0.2239
    0.4887
    0.2874

y =

   -2.6045
    0.9486
s =

    1.0e-009 *

    0.1169
         0
dj =

    0.0683
    0.0000
    0.0000
    0.0000

pobj =

   -0.0932

solstat =

     1
```

Figure 10.4

Reading from Input Files with LMreadf.m

This m-function interfaces with LINDO API to read a model instance in supported file formats and retrieves the problem data into the MATLAB environment.

Run the m-function by invoking the following at the command prompt. This will retrieve the model data of a problem in MPS format into the MATLAB variables specified by LHS arguments of the m-function. Make sure to pass the full path name of the MPS file to the function.

```
>>[c, A, b, l, u, csense, vtype, QCrows, QCvars1, QCvars2, QCcoef] =
LMreadf('c:\lindoapi\samples\mps\testlp.mps');
```

See Appendix B, *MPS File Format*, for general information on MPS files. Also, refer to the description of *LSreadMPSFile*() in Chapter 2, *Function Definitions*, to see different formatting conventions LINDO API supports when reading MPS files.

Column Generation with LMBinPack.m

This function uses a set of LINDO API routines to compute a set-partitioning relaxation to the bin-packing problem based on Dantzig-Wolfe (DW) decomposition.

Suppose n objects with weights w_j, j=1,…,n are given, and the objective is to find the minimum number of bins, each with capacity b, required to pack all n objects. *LMbinpack.m* computes a tight lower bound on the minimum number of bins required using DW-decomposition.

The problem data was represented by a column vector $w = (w_1,w_2,...,w_n)$ and a scalar b. To see how the function works, read the sample bin-packing instance 'bin25_150.mat' that came with the distribution package. This is a small instance with n=25 and b=150. Assuming that your files reside under 'c:\lindoapi', the following commands can be issued to read and run this sample:

```
>> load('bin25_150', 'w', 'b')
>> [E, eb, ec, x, z, how] = LMbinpack(w, b)
```

The output will look like Figure 10.5. The variable z returned by the function is a lower bound on the minimum number of bins required to pack the n objects. The other LHS arguments E, eb, and ec represent the LP data of the set-partitioning formulation of the bin-packing problem.

```
» [E, eb, ec, x, z, how] = LMbinpack(w, b);
      Num cols       Obj of DW    Reduced cost
      generated      relaxation   of new column
      ---------      ----------   -------------
              5          12.000           3.000
             10          12.000           3.588
             15          12.000           9.500
             20          10.879           1.672
             25          10.095           0.429
             30           9.534           0.397
             35           9.100           0.100
             40           9.071           0.071
             45           9.012           0.118
             50           8.976           0.088
             55           8.922           0.047
             55           8.909           0.000

Elapsed time  =      3.324 secs
Minimum bins >=      8.909
```

Figure 10.5

To solve the relaxed set-partitioning formulation as an integer problem, try using *LMsolvem.m* by entering:

```
>> csense = []; vtype = [III…I];
>> l=[]; u = [];
>> [x, y, s, dj, obj, solstat] = LMsolvem(E, eb, ec,csense,l,u,vtype)
```

Chapter 12: An Interface to Ox

Introduction

Ox is an object-oriented programming environment equipped with a comprehensive set of statistical and mathematical functions. In Ox, matrices can be used in expressions with references to their symbolic names providing a particularly attractive medium for modeling and solving optimization problems. Ox's versatile matrix manipulation functions allow users to develop special purpose optimization algorithms quickly and efficiently.

OxLINDO extends the standard capabilities of Ox to include an optimization toolbox by providing an interface to LINDO API's powerful optimizers. In particular, this interface provides Ox users the ability to call LINDO API's functions the same way they call native Ox functions. This offers greater flexibility in developing higher-level Ox routines that can set up and solve different kinds of large-scale optimization problems, testing new algorithmic ideas or expressing new solution techniques.

This release of the interface works with Ox Version 3.x and later. The precompiled binary for OxLINDO can be found in the `\lindoapi\ox` folder. For more information on Ox see http://www.nuff.ox.ac.uk/users/doornik/.

Setting up Ox Interface

For the Windows platform, follow the instructions below to set up the interface. For other platforms, modify the steps accordingly. It is assumed that your LINDO API installation folder is `C:\Lindoapi`.

1. Locate the Ox installation folder on your machine. In a typical Windows installation, it is `C:\Program Files\Ox`.
2. Copy `C:\Lindoapi\Ox` folder to `C:\Program Files\Ox\Packages\Lindoapi\Ox` folder.
3. Copy `C:\Lindoapi\License` folder to `C:\Program Files\Ox\Packages\ Lindoapi\License` folder.
4. Start an Ox session and try out some of the samples located at `C:\lindoapi\samples\ox`.

Calling Conventions

The interface supports all available functions in LINDO API. Because the syntax of Ox's programming language is very similar to the C language, it follows the calling conventions given in Chapter 2 very closely.

Besides the interface functions making native LINDO API calls, OxLINDO has two specific helper functions, that facilitate environment creation and error checking:

1. `OxLScreateEnv( );`
   Check the license and create a LINDO environment. If successful, return an integer referring to a LINDO API environment variable. If unsuccessful, a zero value is returned.

2. `LSerrorCheck(const penv, const nerrorcode);`
   Check the returned error code. If it is nonzero then display the error message associated with specified error code, otherwise do nothing.

These functions are provided for user's convenience and their source codes are available in `oxlindo.ox` file under `\lindoapi\ox` directory. The following code fragment illustrates how these functions are used in a typical Ox optimization session.

```
{
   /* a reference to an instance of the LINDO API environment */
   decl pEnv;

   /* a reference to an instance of the LINDO API model       */
   decl pModel;

  /* Step 1: Create a LINDO environment.                      */
   pEnv = OxLScreateEnv();

   /* Step 2: Create a model in the environment.               */
   pModel = LScreateModel ( pEnv, &nErrorCode);
   LSerrorCheck(pEnv, nErrorCode);
}
```

The following table summarizes the rules for converting a C type into an equivalent Ox type.

C input type	**C description**	**Ox equivalent**
pLSenv	Pointer to Structure	Integer (created with LScreateEnv)
pLSmodel	Pointer to Structure	Integer (created with LScreateModel)
Int	Integer	Integer
double	Double	Double
Int *	Integer vector	Row vector
double *	Double vector	Row vector
char *	Character string	String
char **	Character string array	Array of strings
void *	Pointer to double or integer	Integer or double (LSget..., LSset...)
void *	Pointer to void	Not used (substitute 0 as argument)
NULL	Macro for Null or zero	<>
C output type	**C description**	**Ox equivalent**
Int *	Pointer to integer	Address of variable
Int *	Pointer to integer vector	Address of variable
double *	Pointer to double	Address of variable
double *	Pointer to double vector	Address of variable
char *	Pointer to characters	Address of variable
void *	Pointer to double or integer	Integer or double (LSget..., LSset...)
void *	Pointer to void	Not used (substitute 0 as argument)

Table 10.1 Conversion from C types to Ox

Recall from Chapter 2 that some LINDO API functions accept NULL (in C-style) as one or more of their arguments. In Ox, the <> symbol should replace NULL when necessary in calling such functions. Do not confuse the <> symbol with the <0> expression. The latter corresponds to a constant 1x1 matrix that has a zero value and it cannot substitute the NULL value.

In model or solution access routines, the output arguments should be prefixed with the C-style *address-of* operator "&". This tells Ox that the associated argument is an output argument and ensures that the correct calling convention is used when communicating with LINDO API. For instance, in the following code fragment written in Ox, the output argument MipObj of LSgetInfo is prefixed with "&" operator.

```
{
   decl MipObj;
   decl adX;
   decl nErrorCode;

  /* Retrieve the MIP objective value */
   nErrorCode = LSgetInfo(pModel, LS_DINFO_MIP_OBJ, &MipObj);
   LSerrorCheck(pEnv, nErrorCode);

  /* Retrieve the MIP solution */
   LSgetMIPPrimalSolution( pModel, &adX) ;
   LSerrorCheck(pEnv, nErrorCode);
}
```

Example. Portfolio Selection with Restrictions on the Number of Assets Invested

In the following example, we illustrate how these rules are applied in writing up an equivalent model in Ox to the portfolio selection problem given in Chapter 5. The source codes in C and Ox languages are located under C:\lindoapi\samples\c\port and C:\lindoapi\samples\ox\ folders, respectively.

```
/* port.ox
########################################################################
#                       LINDO-API
#                     Sample Programs
#                   Copyright (c) 2007 by LINDO Systems, Inc
#
#         LINDO Systems, Inc.           312.988.7422
#         1415 North Dayton St.         info@lindo.com
#         Chicago, IL 60622             http://www.lindo.com
########################################################################
  File   : port.ox
  Purpose: Solve a quadratic mixed integer programming problem.
     Model  : Portfolio Selection Problem with a Restriction on
              the Number of Assets

              MINIMIZE   0.5 w'Q w
              s.t.   sum_i  w(i)              =  1
                     sum_i  r(i)w(i)         >=  R
                     for_i  w(i) - u(i) x(i) <=  0   i=1...n
                     sum_i  x(i)             <=  K
                     for_i  x(i) are binary          i=1...n

              where
              r(i)  : return on asset i.
              u(i)  : an upper bound on the proportion of total budget
                      that could be invested on asset i.
              Q(i,j): covariance between the returns of i^th and j^th
                      assets.
              K     : max number of assets allowed in the portfolio
              w(i)  : proportion of total budget invested on asset i
              x(i)  : a 0-1 indicator if asset i is invested on.
```

```
  Data:
  Covariance Matrix:
                A1      A2      A3      A4      A5      A6      A7
      A1 [  1.00    0.11    0.04    0.02    0.08    0.03    0.10 ]
      A2 [  0.11    1.00    0.21    0.13    0.43    0.14    0.54 ]
      A3 [  0.04    0.21    1.00    0.05    0.16    0.05    0.20 ]
 Q =  A4 [  0.02    0.13    0.05    1.00    0.10    0.03    0.12 ]
      A5 [  0.08    0.43    0.16    0.10    1.00    0.10    0.40 ]
      A6 [  0.03    0.14    0.05    0.03    0.10    1.00    0.12 ]
      A7 [  0.10    0.54    0.20    0.12    0.40    0.12    1.00 ]

  Returns Vector:

                A1      A2      A3      A4      A5      A6      A7
   r =    [  0.14    0.77    0.28    0.17    0.56    0.18    0.70 ]

  Maximum Proportion of Total Budget to be Invested on Assets
                A1      A2      A3      A4      A5      A6      A7
   u =    [  0.04    0.56    0.37    0.32    0.52    0.38    0.25 ]

  Target Return:
  R = 0.30

  Maximum Number of Assets:
  K = 3
*/
#include <oxstd.h>
/* LINDO API header file is located under lindoapi\ox */
#import <packages/lindoapi/ox/oxlindo>
/* main entry point */
main()
{
   decl nErrorCode;
/* Number of constraints */
   decl nM = 10;
/* Number of assets (7) plus number of indicator variables (7) */
   decl nN = 14;
/* declare an instance of the LINDO environment object */
   decl pEnv;
/* declare an instance of the LINDO model object */
   decl pModel;
  /*********************************************************
   * Step 1:Create a LINDO environment.MY_LICENSE_KEY in
   * lndapi60.lic must be defined using the key shipped with
   * your software.
   *********************************************************/
   pEnv = OxLScreateEnv();
  /*********************************************************
   * Step 2: Create a model in the environment.
   *********************************************************/
   pModel = LScreateModel ( pEnv, &nErrorCode);
   LSerrorCheck(pEnv, nErrorCode);
   {
  /*********************************************************
   * Step 3: Specify and load the LP portion of the model.
   *********************************************************/
     /* The maximum number of assets allowed in a portfolio */
      decl K = 3;
```

```
    /* The target return */
     decl R = 0.30;
    /* The direction of optimization */
     decl objsense = LS_MIN;
     /* The objective's constant term */
     decl objconst = 0.;
     /* There are no linear components in the objective function.*/
     decl c =       < 0., 0., 0., 0., 0., 0.,0.,
                      0., 0., 0., 0., 0., 0.,0.>;
     /* The right-hand sides of the constraints */
     decl rhs = 1.0 ~ R ~ 0. ~ 0. ~ 0. ~ 0. ~ 0. ~ 0. ~ K;
     /* The constraint types */
     decl contype = "EGLLLLLLLL";
     /* The number of nonzeros in the constraint matrix */
     decl Anz = 35;
     /* The indices of the first nonzero in each column */
     decl Abegcol =    < 0,  3,  6,  9, 12, 15, 18,
                        21, 23, 25, 27, 29, 31, 33> ~ Anz;
     /* The length of each column. Since we aren't leaving
      * any blanks in our matrix, we can set this to NULL */
     decl Alencol = <>;
     /* The nonzero coefficients */
     decl A =       <  1.00, 0.14, 1.00,
                       1.00, 0.77, 1.00,
                       1.00, 0.28, 1.00,
                       1.00, 0.17, 1.00,
                       1.00, 0.56, 1.00,
                       1.00, 0.18, 1.00,
                       1.00, 0.70, 1.00,
                       -0.04, 1.00,
                       -0.56, 1.00,
                       -0.37, 1.00,
                       -0.32, 1.00,
                       -0.52, 1.00,
                       -0.38, 1.00,
                       -0.25, 1.00 >;
     /* The row indices of the nonzero coefficients */
     decl Arowndx =    < 0, 1, 2, 0, 1, 3, 0, 1, 4, 0, 1, 5,
                         0, 1, 6, 0, 1, 7, 0, 1, 8, 2, 9, 3,
                         9, 4, 9, 5, 9, 6, 9, 7, 9, 8, 9    >;
     /* By default, all variables have a lower bound of zero
      * and an upper bound of infinity. Therefore pass NULL
      * pointers in order to use these default values. */
     decl lb = <>, ub = <>;
/*********************************************************
 * Step 4: Specify and load the quadratic matrix
 *********************************************************/
 /* The number of nonzeros in the quadratic matrix */
    decl Qnz = 28;
 /* The nonzero coefficients in the Q-matrix */
 decl Q =       < 1.00,  0.11,  0.04,  0.02,  0.08,  0.03,  0.10,
                  1.00,  0.21,  0.13,  0.43,  0.14,  0.54,
                  1.00,  0.05,  0.16,  0.05,  0.20,
                  1.00,  0.10,  0.03,  0.12,
                  1.00,  0.10,  0.40,
                  1.00,  0.12,
```

```
                    1.00 >;
    /* The row indices of the nonzero coefficients in the Q-matrix*/
    decl  Qrowndx =    < -1, -1, -1, -1, -1, -1, -1,
                         -1, -1, -1, -1, -1, -1,
                         -1, -1, -1, -1, -1,
                         -1, -1, -1, -1,
                         -1, -1, -1,
                         -1, -1,
                         -1 >;
    /* The indices of the first nonzero in each column in the Q-
matrix */
    decl Qcolndx1 =    <  0, 1, 2, 3, 4, 5, 6,
                          1, 2, 3, 4, 5, 6,
                          2, 3, 4, 5, 6,
                          3, 4, 5, 6,
                          4, 5, 6,
                          5, 6,
                          6 >;
    decl Qcolndx2 =    <  0, 0, 0, 0, 0, 0, 0,
                          1, 1, 1, 1, 1, 1,
                          2, 2, 2, 2, 2,
                          3, 3, 3, 3,
                          4, 4, 4,
                          5, 5,
                          6 >;
    /* Pass the linear portion of the data to problem structure
     * by a call to LSloadLPData() */
     nErrorCode = LSloadLPData( pModel, nM, nN, objsense, objconst,
                                c, rhs, contype,
                                Anz, Abegcol, Alencol, A, Arowndx,
                                lb, ub);
     LSerrorCheck(pEnv, nErrorCode);
    /* Pass the quadratic portion of the data to problem structure
     * by a call to LSloadQCData()  */
     nErrorCode = LSloadQCData(pModel, Qnz, Qrowndx,
                               Qcolndx1, Qcolndx2, Q);
     LSerrorCheck(pEnv, nErrorCode);
    /* Pass the integrality restriction to problem structure
     * by a call to LSloadVarData()  */
     {
       decl vartype =  "CCCCCCC"    /* w(j) */
                       "BBBBBBB"  ; /* x(j) */
       nErrorCode = LSloadVarType(pModel, vartype);
       LSerrorCheck(pEnv, nErrorCode);
     }
   }
  /*********************************************************
   * Step 5: Perform the optimization using the MIP solver
   *********************************************************/
   decl nStatus;
   nErrorCode = LSsolveMIP( pModel, &nStatus);
   LSerrorCheck(pEnv, nErrorCode);
   {
  /*********************************************************
   * Step 6: Retrieve the solution
   *********************************************************/
```

```
        decl i;
        decl x, MipObj;
        /* Get the value of the objective and solution */
        nErrorCode = LSgetInfo(pModel, LS_DINFO_MIP_OBJ, &MipObj);
        LSerrorCheck(pEnv, nErrorCode);

        LSgetMIPPrimalSolution( pModel, &x) ;
        LSerrorCheck(pEnv, nErrorCode);
        println("*** Optimal Portfolio Objective = ", MipObj);
        for (i = 0; i < nN/2; i++)
          println( "Invest ", "%5.2f", 100*x[i], " percent of total
budget in asset ",
                   i+1 );
        print("\n");
    }
   /*********************************************************
    * Step 7: Delete the LINDO environment
    *********************************************************/
    nErrorCode = LSdeleteEnv( &pEnv);
} /*main*/
```

After running this program with Ox's console version, we obtain the output depicted in Figure 11.1.

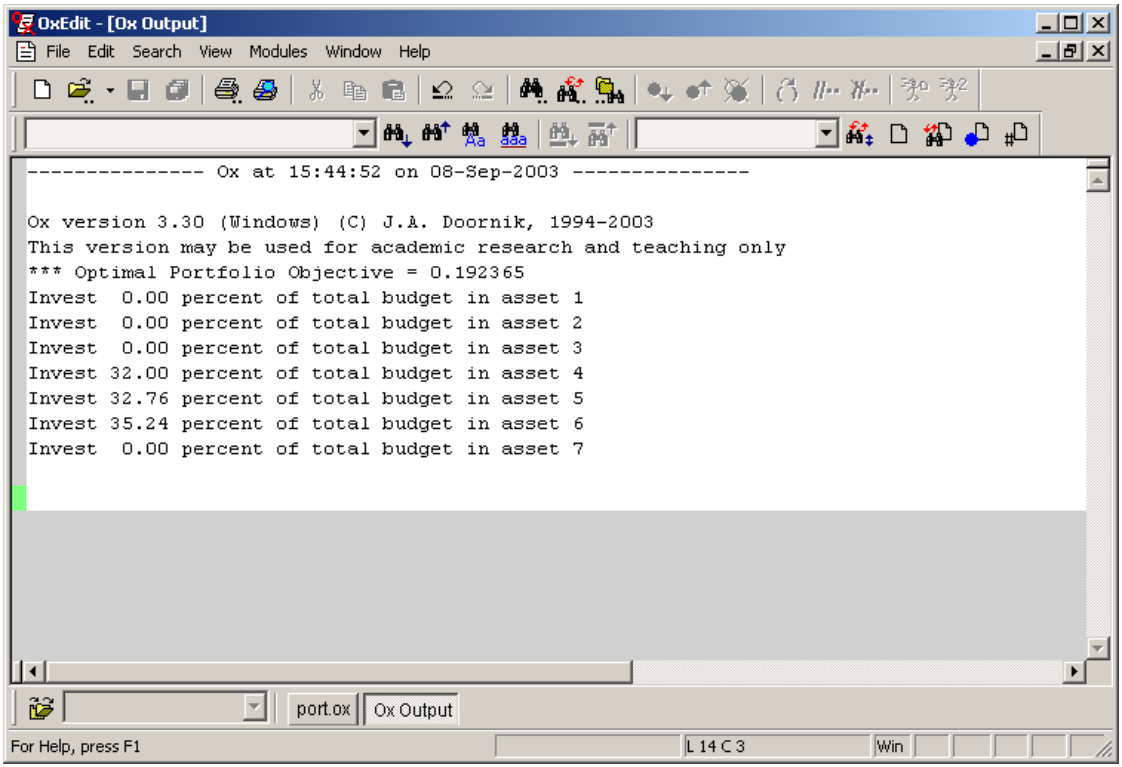

OxEdit - [Ox Output]
File Edit Search View Modules Window Help
---------- Ox at 15:44:52 on 08-Sep-2003 ----------
Ox version 3.30 (Windows) (C) J.A. Doornik, 1994-2003
This version may be used for academic research and teaching only
*** Optimal Portfolio Objective = 0.192365
Invest 0.00 percent of total budget in asset 1
Invest 0.00 percent of total budget in asset 2
Invest 0.00 percent of total budget in asset 3
Invest 32.00 percent of total budget in asset 4
Invest 32.76 percent of total budget in asset 5
Invest 35.24 percent of total budget in asset 6
Invest 0.00 percent of total budget in asset 7
port.ox
Ox Output
For Help, press F1
L 14 C 3
Win

Figure 11.1 Output for Port.ox

Appendix A: Error Codes

Below is a listing of the various error codes that are returned by LINDO API along with a brief description of the error condition and possible remedies. These codes are defined in the header files (e.g. lindo.h) under lindoapi/include directory.

LSERR_BAD_CONSTRAINT_TYPE
Constraint types are expected to be 'G', 'L', 'E', or 'N' corresponding to greater-than-or-equal-to, less-than-or-equal-to, equal-to, and neutral. Correct and retry.

LSERR_BAD_DECOMPOSITION_TYPE
The specified decomposition type is invalid.

LSERR_BAD_LICENSE_FILE
The specified license file does not exist or contains a corrupt license key.

LSERR_BAD_MODEL
There is an error in your formulation. Correct and retry.

LSERR_BAD_MPI_FILE
LINDO API was unable to parse your MPI file for some reason. Check to be sure that the file format follows the rules of the MPI file format and the expressions representing the linear or nonlinear relationships conform to the postfix notation.

LSERR_BAD_MPS_FILE
LINDO API was unable to parse your MPS file for some reason. Check to be sure that the file is truly an MPS file. Review the MPS file format in Appendix B, *MPS File Format*, to see that your file conforms. Try reading the file as an unformatted MPS file.

LSERR_BAD_OBJECTIVE_SENSE
Your objective sense argument was not correctly specified.

LSERR_BAD_SOLVER_TYPE
You've requested an incorrect solver type. Please make sure you have specified one from the supported list of solvers.

LSERR_BAD_VARIABLE_TYPE
The specified variable type is invalid.

LSERR_BASIS_BOUND_MISMATCH
The specified value for basis status does not match to the upper or lower bound the variable can attain.

LSERR_BASIS_COL_STATUS
The specified basis status for a column is invalid.

LSERR_BASIS_INVALID
The given basis is invalid.

LSERR_BASIS_ROW_STATUS
The specified basis status for a constraint's slack/surplus is invalid.

LSERR_BLOCK_OF_BLOCK
The specified model is already a block of a decomposed model.

LSERR_BOUND_OUT_OF_RANGE
The input values fall out side allowed range. E.g. a negative value was input while expecting a nonnegative value.

LSERR_CANNOT_OPEN_FILE
LINDO API couldn't open a specified file. Check the spelling of the file name, be sure that the file exists, and make sure you have read access to the file.

LSERR_CHECKSUM
A checksum operation has failed during license checking.

LSERR_COL_BEGIN_INDEX
The index vector that mark the beginning of structural columns in three (or four) vector representation of the underlying model is invalid.

LSERR_COL_INDEX_OUT_OF_RANGE
The specified column index is out of range for the underlying model.

LSERR_COL_NONZCOUNT
The number of nonzeros in one or more columns specified is invalid or inconsistent with other input vectors.

LSERR_ERRMSG_FILE_NOT_FOUND
The specified file was not found.

LSERR_ERROR_IN_INPUT
There was an error in the input.

LSERR_GOP_BRANCH_LIMIT
The GOP solver has reached the branch limit in branch and bound before solving to optimality.

LSERR_GOP_FUNC_NOT_SUPPORTED
The specified function is not supported with GOP solver

LSERR_ILLEGAL_NULL_POINTER
LINDO API was expecting a pointer as an argument, but found NULL instead.

LSERR_INDEX_DUPLICATE
The specified index set contains duplicate index values.

LSERR_INDEX_OUT_OF_RANGE
The specified index is out of range.

LSERR_INSTRUCT_NOT_LOADED
The instruction list has not yet been loaded into the model specified by a *pLSmodel* type pointer.

LSERR_INTERNAL_ERROR
An unanticipated internal error has occurred. Please report this problem to LINDO Systems Technical Support.

LSERR_INFO_NOT_AVAILABLE
You have posed a query to LINDO API for which no information is available.

LSERR_INVALID_ERRORCODE
The error code inquired about is invalid.

LSERR_ITER_LIMIT
The solver reached the iteration limit before solving to optimality.

LSERR_LAST_ERROR
This error code marks the last valid error code in LINDO API and is for internal use only.

LSERR_MIP_BRANCH_LIMIT
The solver has reached the branch limit in branch and bound before solving to optimality.

LSERR_MODEL_ALREADY_LOADED
The problem data has already been loaded into the model specified by a *pLSmodel* type pointer.

LSERR_MODEL_NOT_LINEAR
The underlying model is not linear.

LSERR_MODEL_NOT_LOADED
The problem data has not yet been loaded into the model specified by a *pLSmodel* type pointer.

LSERR_NO_ERROR
The LINDO API function called has terminated without any errors.

LSERR_NO_LICENSE_FILE
No license file that contains a valid license could be found on the system.

LSERR_NO_METHOD_LICENSE
Your license key doesn't allow for the solver method you've chosen. To check the capacity of your version, call *LSgetModelIntParameter*() with license information access macros. Try a different solver method or upgrade your license to include the desired method.

LSERR_NO_VALID_LICENSE
The license key passed to *LScreateEnv*() was not valid. Please check that you have correctly typed in your license key, preserving capitalization and including all hyphens.

LSERR_NOT_CONVEX
The underlying model is not convex. This implies that the model could not be solved using the standard barrier solver.

LSERR_NOT_SUPPORTED
You have tried to use a feature that is not currently supported.

LSERR_NUMERIC_INSTABILITY
The solver encountered a numeric error and was unable to continue. Please report this problem to LINDO Systems Technical Support.

LSERR_OLD_LICENSE
The license is valid for an older version.

LSERR_OUT_OF_MEMORY
You don't have adequate memory for the operation. Add more RAM and/or free disk space to allow the operating system more swap space.

LSERR_PARAMETER_OUT_OF_RANGE
The specified parameter was out of range.

LSERR_ROW_INDEX_OUT_OF_RANGE
The specified row index is out of range for the underlying model.

LSERR_STEP_TOO_SMALL
The solver halted because of failing to take sufficiently large steps to the solution set.

LSERR_TIME_LIMIT
The solver reached the time limit before solving to optimality.

LSERR_TOO_SMALL_LICENSE
Your license key doesn't allow for enough capacity to solve the model you've built. To check the capacity of your version, call *LSgetModelIntParameter*() with license information access macros. You'll need to reduce the size of your model or upgrade to a larger license.

LSERR_TOTAL_NONZCOUNT
The total number of nonzeros specified is invalid or inconsistent with other input.

LSERR_TRUNCATED_NAME_DATA
The solver exported the specified model in a portable file format, however, some variables or constraints had very long names which have been truncated to a fixed length while exporting.

LSERR_UNABLE_TO_SET_PARAM
The parameter you are attempting to set is not user configurable.

LSERR_USER_FUNCTION_NOT_FOUND
Model contains user function that is not supplied.

LSERR_USER_INTERRUPT
The solver was interrupted by the user's callback function.

LSERR_VARIABLE_NOT_FOUND
The specified variable was not found in the model.

LSERR_DATA_TERM_EXIST
The row already has a quadratic (or nonlinear) term loaded.

LSERR_NOT_SORTED_ORDER
The index vector is required to be sorted but it is not.

LSERR_INST_MISS_ELEMENTS
Instruction list has incorrect numbers of elements.

LSERR_INST_TOO_SHORT
Instruction list has too short a length.

LSERR_INST_INVALID_BOUND
Instruction list has conflicting variable bounds. For example, the lower bound is greater than the upper bound.

LSERR_INST_SYNTAX_ERROR
Instruction list contains at least one syntax error.

LSERR_LAST_ERROR
Marker for the last error code. Internal use only.

LSERR_BAD_SMPS_CORE_FILE
Core MPS file/model has an error.

LSERR_BAD_SMPS_TIME_FILE
Time file/model has an error.

LSERR_BAD_SMPS_STOC_FILE
Stoc file/model has an error.

LSERR_BAD_SMPI_CORE_FILE
Core MPI file/model has an error.

LSERR_BAD_SMPI_STOC_FILE
Stoc file associated with Core MPI file has an error.

LSERR_CANNOT_OPEN_CORE_FILE
Unable to open Core file.

LSERR_CANNOT_OPEN_TIME_FILE
Unable to open Time file.

LSERR_CANNOT_OPEN_STOC_FILE
Unable to open Stoc file.

LSERR_STOC_MODEL_NOT_LOADED
Stochastic model/data has not been loaded yet.

LSERR_STOC_SPAR_NOT_FOUND
Stochastic parameter specified in Stoc file has not been found .

LSERR_TIME_SPAR_NOT_FOUND
Stochastic parameter specified in Time file has not been found .

LSERR_SCEN_INDEX_OUT_OF_SEQUENCE
Specified scenario index is out of sequence.

LSERR_STOC_MODEL_ALREADY_PARSED
Stochastic model/data has already been loaded.

LSERR_STOC_INVALID_SCENARIO_CDF
Specified scenario CDF is invalid, e.g. scenario probabilities don't sum to 1.0

LSERR_CORE_SPAR_NOT_FOUND
No stochastic parameters was found in the Core file.

LSERR_CORE_SPAR_COUNT_MISMATCH
Number of stochastic parameters found in Core file don't match to that of Time file.

LSERR_CORE_INVALID_SPAR_INDEX
Specified stochastic parameter index is invalid.

LSERR_TIME_SPAR_NOT_EXPECTED
A stochastic parameter was not expected in Time file.

LSERR_TIME_SPAR_COUNT_MISMATCH
Number of stochastic parameters found in Time file don't match to that of Stoc file.

LSERR_CORE_SPAR_VALUE_NOT_FOUND
Specified stochastic parameter doesn't have a valid outcome value.

LSERR_INFO_UNAVAILABLE
Requested information is unavailable.

LSERR_STOC_MISSING_BNDNAME
Core file doesn't have a valid bound name tag.

LSERR_STOC_MISSING_OBJNAME
Core file doesn't have a valid objective name tag.

LSERR_STOC_MISSING_RHSNAME
Core file doesn't have a valid right-hand-side name tag.

LSERR_STOC_MISSING_RNGNAME
Core file doesn't have a valid range name tag.

LSERR_MISSING_TOKEN_NAME
Stoc file doesn't have an expected token name.

LSERR_MISSING_TOKEN_ROOT
Stoc file doesn't have a 'ROOT' token to specify a root scenario.

LSERR_STOC_NODE_UNBOUNDED
Node model is unexpectedly unbounded.

LSERR_STOC_NODE_INFEASIBLE
Node model is unexpectedly infeasible.

LSERR_STOC_TOO_MANY_SCENARIOS
Stochastic model has too many scenarios to solve with specified solver.

LSERR_STOC_BAD_PRECISION
One or more node-models have irrecoverable numerical problems.

LSERR_CORE_BAD_STRUCTURE
Specified aggregation structure is not compatible with model's stage structure.

LSERR_STOC_NULL_EVENT_TREE
Event tree is either not initialized yet or was too big to create.

LSERR_CORE_BAD_STAGE_INDEX
Specified stage index is invalid.

LSERR_STOC_BAD_ALGORITHM
Specified algorithm/method is invalid or not supported.

LSERR_CORE_BAD_NUMSTAGES
Specified number of stages in Core model is invalid.

LSERR_TIME_BAD_TEMPORAL_ORDER
Underlying model has an invalid temporal order.

LSERR_TIME_BAD_NUMSTAGES
Number of stages specified in Time structure is invalid.

LSERR_CORE_TIME_MISMATCH
Core and Time data are inconsistent.

LSERR_STOC_INVALID_CDF
Specified stochastic structure has an invalid CDF.

LSERR_BAD_DISTRIBUTION_TYPE
Specified distribution type is invalid or not supported.

LSERR_DIST_SCALE_OUT_OF_RANGE
Scale parameter for specified distribution is out of range.

LSERR_DIST_SHAPE_OUT_OF_RANGE
Shape parameter for specified distribution is out of range.

LSERR_DIST_INVALID_PROBABILITY
Specified probabability value is invalid.

LSERR_DIST_NO_DERIVATIVE
Derivative information is unavailable.

LSERR_DIST_INVALID_SD
Specified standard deviation is invalid.

LSERR_DIST_INVALID_X
Specified value is invalid.

LSERR_DIST_INVALID_PARAMS
Specified parameters are invalid for the given distribution.

LSERR_DIST_ROOTER_ITERLIM
Iteration limit has been reached during a root finding operation.

LSERR_ARRAY_OUT_OF_BOUNDS
Given array is out of bounds.

LSERR_DIST_NO_PDF_LIMIT
Limiting PDF does not exist

LSERR_RG_NOT_SET
A random number generator is not set.

LSERR_DIST_TRUNCATED
Distribution function value was truncated during calculations.

LSERR_STOC_MISSING_PARAM_TOKEN
Stoc file has a parameter value missing.

LSERR_DIST_INVALID_NUMPARAM
Distribution has invalid number of parameters.

LSERR_CORE_NOT_IN_TEMPORAL_ORDER
Core file/model is not in temporal order.

LSERR_STOC_INVALID_SAMPLE_SIZE
Specified sample size is invalid.

LSERR_STOC_NOT_DISCRETE
Node probability cannot be computed due to presence of continuous stochastic parameters.

LSERR_STOC_SCENARIO_LIMIT
Event tree exceeds the maximum number of scenarios allowed to attempt an exact solution.

LSERR_DIST_BAD_CORRELATION_TYPE
Specified correlation type is invalid.

LSERR_TIME_NUMSTAGES_NOT_SET
Number of stages in the model is not set yet.

LSERR_STOC_SAMPLE_ALREADY_LOADED
Model already contains a sampled tree

LSERR_STOC_EVENTS_NOT_LOADED
Stochastic events are not loaded yet

LSERR_STOC_TREE_ALREADY_INIT
Stochastic tree already initialized

LSERR_RG_SEED_NOT_SET
Random number generator seed not initialized

LSERR_STOC_OUT_OF_SAMPLE_POINTS
All sample points in the sample has been used. Resampling may be required.

LSERR_STOC_SCENARIO_SAMPLING_NOT_SUPPORTED
All sample points in the sample has been used. Resampling may be required.

LSERR_STOC_SAMPLE_NOT_GENERATED
Sample points are not yet generated for a stochastic parameter.

LSERR_STOC_SAMPLE_ALREADY_GENERATED
Sample points are already generated for a stochastic parameter.

LSERR_STOC_SAMPLE_SIZE_TOO_SMALL
Sample sizes selected are too small.

LSERR_RG_ALREADY_SET
A random number generator is already set.

LSERR_STOC_BLOCK_SAMPLING_NOT_SUPPORTED
Sampling is not allowed for block/joint distributions.

LSERR_EMPTY_ROW_STAGE
No rows were assigned to one of the stages.

LSERR_EMPTY_COL_STAGE
No columns were assigned to one of the stages.

LSERR_STOC_CONFLICTING_SAMP_SIZES
Default sample sizes per stoc.pars and stage are in conflict.

LSERR_STOC_EMPTY_SCENARIO_DATA
Empty scenario data

LSERR_STOC_CORRELATION_NOT_INDUCED
A correlation structure has not been induced yet

LSERR_STOC_PDF_TABLE_NOT_LOADED
A discrete PDF table has not been loaded

Appendix B: MPS File Format

This appendix describes the file format that can be read with *LSreadMPSFile*(), or written with *LSwriteMPSFile*(). The MPS format for describing an LP or a quadratic program is a format commonly used in industry. It is a text file format, so one of the reasons for using it is to move an LP/IP model from one machine to another machine of a different type or manufacturer. It is not a very compact format (i.e., MPS format files tend to be large and wasteful of space).

Every MPS file has at least the two sections:

ROWS	(List the row names and their type: L, E, G, or N)
COLUMNS	(List by column, nonzero elements in objective and constraints)

The optional sections in an MPS file are :

RHS	(Specify nonzero right hand sides for constraints.)
BOUNDS	(Specify bounds on variables.)
RANGES	(Specify the bounds on a RHS.)
QMATRIX	(Specify a quadratic portion of a row or the objective function)
QSECTION	(Synonym for QMATRIX)
CSECTION	(Specify second order cone constraints)

Any line with an asterisk (*) in the first position is treated as a comment line and is disregarded.

LINDO API understands the most commonly used features of the MPS format subject to:

1. Leading blanks in variable and row names are disregarded. All other characters, including embedded blanks, are allowed.
2. Only one free row (type *N* row) is retained from the ROWS section after input is complete, specifically the one selected as the objective.
3. Only one BOUNDS set is recognized in the BOUNDS section. Recognized bound types are:

UP	(upper bound)
LO	(lower bound)
FR	(free variable)
FX	(fixed variable)
BV	(bivalent variable, i.e., 0/1 variables)
UI	(upper-bounded integer variable)
LI	(lower-bounded integer variable)
SC	(semi-continuous variable)

4. Only one RANGES set is recognized in the RANGES section.
6. MODIFY sections are not recognized.
7. SCALE lines are accepted, but have no effect.

Even though embedded blanks are allowed in names in an MPS file, they are not recommended. For example, even though "OK NAME" is an acceptable name for a row in an MPS file, it is not recommended.

Similarly, lowercase names are accepted, but for consistency—also for ease of distinguishing between 1 (one) and l (L)—it is recommended that only uppercase names be used.

To illustrate an MPS format file, consider the following equation style model in LINGO format:

```
[PROFIT] MAX = 500*LEXUS + 1600*CAMARO + 4300* BEETLE +
1800*BMW;
[MIX]      12*LEXUS          -4*BEETLE -  2*BMW  >= 0;
[SPORT]       CAMARO                     + BMW  <= 2000;
[SMALL]                        BEETLE    + BMW  <= 1500;
[TOTAL]     LEXUS + CAMARO + BEETLE   + BMW      <= 3000;
! This lower bound on the SMALL constraint can be represented
    by an entry in the RANGES section of an MPS file;
[SMALLR]                     BEETLE   + BMW      >= 1500-700;
! This upper bound on a variable can be represented by an
    entry in the BOUNDS section of an MPS file;
@BND(0, LEXUS, 250);
```

The equivalent MPS file looks like:

```
NAME          CAFEMODL
ROWS
 N  PROFIT
 G  MIX
 L  SPORT
 L  SMALL
 L  TOTAL
COLUMNS
    LEXUS     PROFIT          -500
    LEXUS     MIX               12
    LEXUS     TOTAL              1
    CAMARO    PROFIT         -1600
    CAMARO    SPORT              1
    CAMARO    TOTAL              1
    BEETLE    PROFIT         -4300
    BEETLE    TOTAL              1
    BEETLE    MIX               -4
    BEETLE    SMALL              1
    BMW       PROFIT         -1800
    BMW       MIX               -2
    BMW       TOTAL              1
    BMW       SMALL              1
    BMW       SPORT              1
RHS
    RHS1      SPORT           2000
    RHS1      SMALL           1500
    RHS1      TOTAL           3000
RANGES
    ROWRNG1   SMALL            700
BOUNDS
 UP BND1      LEXUS            250
ENDATA
```

Notice that there are two major types of lines in an MPS file: (1) header lines such as ROWS, COLUMNS, RHS, etc., and (2) data lines, which immediately follow each header line. The fields in a data line are as follows:

Field	Character Position	Contents
1	2 to 3	Row type or bound type
2	5 to 12	Name of column, bound or range
3	15 to 23	Row name
4	25 to 37	Numerical value
5	40 to 47	Row name
6	50 to 62	Numerical value

Two features of an MPS file are worth noting at this point: (1) It is allowed to have several non-constrained rows (i.e., type *N*) any one of which could be the objective and (2) There is nothing in the file to indicate whether it is a MIN or a MAX problem. The default is that it is MIN, so in our example, the signs have been reversed in the MPS file on the coefficients in the MAX objective.

Integer Variables

The standard way of designating integer variables in an MPS file is to place them between 'INTORG', 'INTEND' marker cards in the COLUMNS section. Integer variables may alternatively be designated with either the BV, UI, or LI type in a BOUNDS section. Consider the following model in LINGO equation style.

```
! Example: EXAMINT;
[OBJ] MIN = 38*X1   + 42*X2 + 14*X3 + 28*X4;
[NEED]   12*X1 + 14*X2 + 6*X3 + 12*X4 >=  78;
@GIN(X1); @GIN(X2); @GIN(X3);
@BND(0,X3,2);
@BIN(X4);
```

An MPS file describing the above model is:

```
NAME          EXAMINT
ROWS
 N  OBJ
 G  NEED
COLUMNS
    MYINTS1   'MARKER'                 'INTORG'
    X1  OBJ                38
    X1  NEED               12
    X2  OBJ                42
    X2  NEED               14
    MYINTS1   'MARKER'                 'INTEND'
    X3  OBJ                14
    X3  NEED                6
    X4  OBJ                28
    X4  NEED               12
RHS
```

```
    RHS1      NEED         78
BOUNDS
 UP BND1      X1         9999
 UP BND1      X2         9999
 UI BND1      X3            2
 BV BND1      X4
ENDATA
```

Some software systems assume an upper bound of 1.0 on any variable appearing in an INTORG, INTEND section, so the safe approach is to always explicitly list the intended upper bound of an integer variable in the BOUNDS section.

Semi-continuous Variables

A semi-continuous variable is one that is constrained to be either 0 or strictly positive over a range. Such a semi-continuous variable is indicated by using the SC bound type in the BOUNDS section. The following equation form model illustrates.

```
TITLE SEMICONT;
 [ OBJ] MIN = - 20 * A - 38 * C - 44 * R;
 [ALINE]             A              +  R <= 60;
 [CLINE]                       C +      R <= 50;
 [LABOR]             A  + 2 * C  + 3 * R <= 119;
 @GIN( C); @GIN( R);
 @BND( 0, C, 45); @BND( 0, R, 999);
! Additionally, we want either C = 0, or 35 <= C <= 45;
```

The above model does not enforce the semi-continuous feature on C. In the MPS format you can easily enforce the feature by using the SC bound type in the BOUNDS section. See below.

```
NAME          SEMICONT Illustrate semi-continuous variables
ROWS
 N  OBJ
 L  ALINE
 L  CLINE
 L  LABOR
COLUMNS
    A  OBJ               -20
    A  LABOR               1
    A  ALINE               1
    INT0000B  'MARKER'                 'INTORG'
    C  OBJ               -38
    C  LABOR               2
    C  CLINE               1
    R  OBJ               -44
    R  ALINE               1
    R  LABOR               3
    R  CLINE               1
    INT0000E  'MARKER'                 'INTEND'
RHS
    RHS1      ALINE         60
    RHS1      CLINE         50
    RHS1      LABOR        119
BOUNDS
 SC BND1      C             45
```

```
 LO BND1      C            35
 UP BND1      R           999
* We must have either C = 0 or 35 <= C <= 45
* If the LO bound does not appear for an SC variable
*  then it is assumed to be 1.
* Appearance of both SC and UP for a variable is an error.
ENDATA
```

SOS Sets

SOS(Special Ordered Sets) provide a compact way of specifying multiple choice type conditions. The LINDO API recognizes three types of SOS sets. A set of variables defined to be in an SOS will be constrained in the following ways.

- Type 1: At most one of the variables in the set will be allowed to be nonzero.
- Type 2: At most two variables in the set will be allowed to be nonzero. If two, they must be adjacent.
- Type 3: At most one of the variables in the set will be nonzero. If one, its value must be 1.

Consider the following example.

```
[ OBJ] MIN = -3*X1 -2*X2 -4*X3;
 [ R2]  X1 +  X2 +  X3 <= 5;
 [ R3]  X1             <= 2;
 [ R4]        X2       <= 2;
 [ R5]              X3 <= 2;
```

The following MPS file will cause X1, X2, and X3 to be in a type 1 SOS set.

```
NAME          S3TEST
ROWS
 N  OBJ
 L  R2
 L  R3
 L  R4
 L  R5
COLUMNS
 S1 JUNK      'MARKER'                 'SOSORG'
    X1        OBJ         -3
    X1        R2           1
    X1        R3           1
    X2        OBJ         -2
    X2        R2           1
    X2        R4           1
    X3        OBJ         -4
    X3        R2           1
    X3        R5           1
 S1 JUNK      'MARKER'                 'SOSEND'
RHS
    RHS1      R2           5
    RHS1      R3           2
    RHS1      R4           2
    RHS1      R5           2
ENDATA
```

The optimal solution will be X1 = X2 = 0, X3 = 2.

If you change the S1 to S2 in the MPS file, then the optimal solution will be X1 = 0, X2 = X3 = 2.

If you change the S1 to blanks, e.g., the start marker line is simply:

```
JUNK      'MARKER'                 'SOSORG'
```

then X1, X2, and X3 will be interpreted as a type 3 SOS set and the optimal solution will be:

The optimal solution will be X1 = X2 = 0, X3 = 1.

SOS2 Example

An SOS2 set is an ordered set of variables which are required to satisfy the conditions: a) at most two variables in the set may be nonzero, and b) if two, then they must be adjacent. This feature is useful for modeling piecewise linear continuous curves. The following example illustrates.

```
! Cost of production is a piecewise linear, continuous
function of 4 segments given by the 5 points:
      cost:   0  1500 15500 41500 77500
    volume:   0   100  1100  3100  6100.
We have 3 customers who are willing to buy
at a given price/unit up to a maximum.
Maximize revenues minus cost of production;
   Max = 20*SELL1 + 14*SELL2 + 13*SELL3 - COST;
! How much each customer will buy;
   @BND(0,SELL1,300); @BND(0,SELL2,900); @BND(0,SELL3,2000);
! Wj =weight given to each point on cost curve;
   W0 + W0100      + W1100      + W3100      + W6100= 1;
    100*W0100 + 1100*W1100 + 3100*W3100 + 6100*W6100= VOL;
   1500*W0100 +15500*W1100 +41500*W3100 +77500*W6100= COST;
! If we sell it, we have to make it;
   SELL1 + SELL2 + SELL3 = VOL;
! Additionally, we need the SOS2 condition that at most
   2 W's are > 0, and they must be adjacent;
! Soln: Obj=1900, W3100=0.9666667, W6100= 0.0333333, VOL=
3200;
```

The above model does not enforce the SOS2 feature on W0,…,W6100. An MPS file for this model that enforces the SOS2 condition is:

```
NAME           SOS3EXAM  Illustrate use of SOS2 set
ROWS
 N  OBJ
 E  CNVX
 E  CVOL
 E  CCST
 E  BALN
COLUMNS
    SELL1     OBJ          -20
    SELL1     BALN           1
    SELL2     OBJ          -14
    SELL2     BALN           1
    SELL3     OBJ          -13
    SELL3     BALN           1
```

```
    COST      OBJ            1
    COST      CCST          -1
 S2 SET2      'MARKER'                 'SOSORG'
    W0        CNVX           1
    W0100     CNVX           1
    W0100     CCST        1500
    W0100     CVOL         100
    W1100     CNVX           1
    W1100     CVOL        1100
    W1100     CCST       15500
    W3100     CNVX           1
    W3100     CVOL        3100
    W3100     CCST       41500
    W6100     CNVX           1
    W6100     CVOL        6100
    W6100     CCST       77500
 S2 SET2      'MARKER'                 'SOSEND'
    VOL       CVOL          -1
    VOL       BALN          -1
RHS
    RHS1      CNVX           1
BOUNDS
 UP BND1      SELL1        300
 UP BND1      SELL2        900
 UP BND1      SELL3       2000
ENDATA
```

Quadratic Objective

A quadratic objective function may be input via the MPS format by entering the coefficients of the quadratic function. Consider the following equation form model.

```
[VAR] MIN=
   X1*X1 * 0.01080754 + X1*X2 * 0.01240721 + X1*X3 * 0.01307513
 + X2*X1 * 0.01240721 + X2*X2 * 0.0583917  + X2*X3 * 0.05542639
 + X3*X1 * 0.01307513 + X3*X2 * 0.05542639 + X3*X3 * 0.09422681 ;
 [BUDGET] X1 + X2 + X3 = 1 ;
 [RETURN] 1.0890833 * X1 + 1.213667 * X2 + 1.234583 * X3 >=
1.15 ;
```

A quadratic objective can be described in an MPS file by a QMATRIX section as shown below. The second field `VAR` in `QMATRIX` header must correspond to the objective function name listed in the `ROWS` section.

```
       NAME          PORTQP   Markowitz's portfolio problem
*  [VAR] MIN=
*   X1*X1 * 0.01080754 + X1*X2 * 0.01240721 + X1*X3 * 0.01307513
* + X2*X1 * 0.01240721 + X2*X2 * 0.0583917  + X2*X3 * 0.05542639
* + X3*X1 * 0.01307513 + X3*X2 * 0.05542639 + X3*X3 * 0.09422681 ;
* [BUDGET] X1 + X2 + X3 = 1 ;
* [RETURN] 1.0890833 * X1 + 1.213667 * X2 + 1.234583 * X3 >= 1.15 ;
*
* Input to QP optimizers assume quadratic has been divided by 2.0,
* so when first derivatives are taken the 2's cancel.
ROWS
 N  VAR
 E  BUDGET
 G  RETURN
COLUMNS
    X1        BUDGET              1
    X1        RETURN              1.0890833
    X2        BUDGET              1
    X2        RETURN              1.213667
    X3        BUDGET              1
    X3        RETURN              1.234583
RHS
    rhs       BUDGET              1
    rhs       RETURN              1.15
QMATRIX       VAR
    X1        X1                  0.02161508
    X1        X2                  0.02481442
    X1        X3                  0.02615026
    X2        X1                  0.02481442
    X2        X2                  0.1167834
    X2        X3                  0.11085278
    X3        X1                  0.02615026
    X3        X2                  0.11085278
    X3        X3                  0.18845362
* The upper triangular is input.
ENDATA
```

Quadratic Constraints

A quadratic constraint may be input via the MPS format by entering the coefficients of the quadratic function. Consider the following equation form model.

```
[RETURN] MAX
   = 1.0890833 * X1 + 1.213667 * X2 + 1.234583 * X3 ;
 [VAR]
   X1*X1 * 0.01080754 + X1*X2 * 0.01240721 + X1*X3 * 0.01307513
 + X2*X1 * 0.01240721 + X2*X2 * 0.0583917  + X2*X3 * 0.05542639
 + X3*X1 * 0.01307513 + X3*X2 * 0.05542639 + X3*X3 * 0.09422681 <=
0.02241375 ;
 [BUDGET] X1 + X2 + X3 = 1 ;
```

A quadratic constraint is described in an MPS file by a `QMATRIX` section as shown below. The second field `VAR` in `QMATRIX` header must be the associated constraint name listed in the `ROWS` section.

```
NAME          PORTQPC
ROWS
 N  RETURN
 L  VAR
 E  BUDGET
COLUMNS
    X1        RETURN      -1.0890833
    X1        BUDGET       1
    X2        RETURN      -1.213667
    X2        BUDGET       1
    X3        RETURN      -1.234583
    X3        BUDGET       1
QMATRIX       VAR
    X1        X1           0.02161508
    X1        X2           0.02481442
    X1        X3           0.02615026
    X2        X1           0.02481442
    X2        X2           0.1167834
    X2        X3           0.11085278
    X3        X1           0.02615026
    X3        X2           0.11085278
    X3        X3           0.18845362
RHS
    RHS1      BUDGET       1
    RHS1      VAR          0.02241375
ENDATA
```

The quadratic matrix must be symmetric. If the barrier solver is used, the quadratic matrix must be positive semi-definite.

Second Order Cone Constraints

The LINDO API supports two types of second order cone constraints: a) simple quadratic cones, denoted by `QUAD`, and b) rotated quadratic cones, denoted by `RQUAD`. A simple quadratic cone constraint is of the form:

$$-x_0^2 + x_1^2 + x_2^2 + \ldots + x_n^2 \le 0;$$
$$x_0 \ge 0;$$

A rotated quadratic cone constraint is of the form:

$$-2x_0 x_1 + x_2^2 + x_3^2 + \ldots + x_n^2 \le 0;$$
$$x_0, x_1 \ge 0;$$

Consider the following example of a simple cone constraint in equation form.

```
[OBJ] MIN = -4*X1 - 5*X2 - 6*X3;
[CAP] 8*X1 + 11*X2 + 14*X3 + 1.645*SD <= 34.8;
[S1] SD1 - 2*X1 = 0;
[S2] SD2 - 3*X2 = 0;
[S3] SD3 - 4.1*X3 = 0;
[CONE1] SD1^2 + SD2^2 + SD3^2 - SD^2 <= 0;
  @BND(0,X1,1); @BND(0,X2,1); @BND(0,X3,1);
```

The MPS file describing this model is:

```
NAME          CONE2EX1  Model with a single QUADratic cone
ROWS
 N  OBJ
 L  CAP
 E  S1
 E  S2
 E  S3
COLUMNS
    X1        OBJ         -4
    X1        CAP          8
    X1        S1          -2
    X2        OBJ         -5
    X2        CAP         11
    X2        S2          -3
    X3        OBJ         -6
    X3        CAP         14
    X3        S3          -4.1
    SD        CAP          1.645
    SD1       S1           1
    SD2       S2           1
    SD3       S3           1
RHS
    RHS1      CAP              34.8
BOUNDS
 UP BND1      X1                 1
 UP BND1      X2                 1
 UP BND1      X3                 1
CSECTION      CONE1      0.0           QUAD
    SD
    SD1
    SD2
    SD3
ENDATA
```

We illustrate a rotated quadratic cone constraint with the following model in equation form:

```
[OBJ] MIN = 2*HGT + 1.5*WID
  - 5*RADIUS1 - 4*RADIUS2 - 3.5*RADIUS3;
[TPI1] R1 - 1.77245385*RADIUS1 = 0;
[TPI2] R2 - 1.77245385*RADIUS2 = 0;
[TPI3] R3 - 1.77245385*RADIUS3 = 0;
[WGT1] 3.5*RADIUS1 + 3*RADIUS2 + 2.5*RADIUS3 <=  6;
[WGT2]   4*RADIUS1 + 6*RADIUS2   + 5*RADIUS3 <= 11;
[CONE2] R1^2 + R2^2 + R3^2 - 2*HGT*WID <= 0;
```

The corresponding MPS file is:

```
NAME           CONE2EX2 Rotated cone example
ROWS
 N  OBJ
 E  TPI1
 E  TPI2
 E  TPI3
 L  WGT1
 L  WGT2
COLUMNS
    HGT       OBJ                2
    WID       OBJ              1.5
    RADIUS1   OBJ               -5
    RADIUS1   TPI1      -1.77245385
    RADIUS1   WGT1              3.5
    RADIUS1   WGT2                4
    RADIUS2   OBJ               -4
    RADIUS2   TPI2      -1.77245385
    RADIUS2   WGT1                3
    RADIUS2   WGT2                6
    RADIUS3   OBJ             -3.5
    RADIUS3   TPI3      -1.77245385
    RADIUS3   WGT1              2.5
    RADIUS3   WGT2                5
    R1        TPI1                1
    R2        TPI2                1
    R3        TPI3                1
RHS
    RHS1      WGT1                6
    RHS1      WGT2               11
CSECTION      CONE2            0.0     RQUAD
    HGT
    WID
    R1
    R2
    R3
ENDATA
```

A cone constraint need not be defined in the ROWS section. There are some restrictions on the usage of cone constraints: a) If there are any cone constraints, then there cannot be any quadratic terms, i.e., if a CSECTION appears in a model, then there can be no QMATRIX or QSECTION sections, b) a variable can appear in at most one CSECTION. However, these limitations need not be tight provided that correct formulation is used. For instance, general convex quadratically constrained models can be easily cast as conic models by simple change of variables. Similarly, by using auxiliary variables, arbitrary conic constraints can be formulated with where any variable appears in at most one CSECTION.

Appendix C: LINDO File Format

The MPS file format is a column-oriented format. If a row-oriented format is more convenient, then the LINDO file format is of interest. This section details the syntax required in a model imported from a text file with *LSreadLINDOFile*(). The list of rules is rather short and easy to learn.

Flow of Control

The objective function must always be at the start of the model and is initiated with any of the following keywords:

```
MAX                 MIN
MAXIMIZE            MINIMIZE
MAXIMISE            MINIMISE
```

The end of the objective function and the beginning of the constraints are signified with any of the following keywords:

```
SUBJECT TO
SUCH THAT
S.T.
ST
```

The end of the constraints is signified with the word END.

Formatting

Variable names are limited to eight characters. Names must begin with an alphabetic character (A to Z), which may then be followed by up to seven additional characters. These additional characters may include anything with the exception of the following: !) + - = < >. As an example, the following names are valid:

```
XYZ          MY_VAR          A12          SHIP.LA
```

whereas the following are not:

```
THISONEISTOOLONG          A-HYPHEN          1INFRONT
```

The first example contains more than eight characters, the second contains a forbidden hyphen, and the last example does not begin with an alphabetic character.

You may, optionally, name constraints in a model. Constraint names must follow the same conventions as variable names. To name a constraint, you must start the constraint with its name terminated with a right parenthesis. After the right parenthesis, you enter the constraint as before. As an example, the following constraint is given the name *XBOUND*:

```
XBOUND) X < 10
```

Only five operators are recognized: plus (+), minus (-), greater than (>), less than (<), and equals (=). When you enter the strict inequality operators greater than (>) and less than (<), they will be interpreted as the loose inequality operators greater-than-or-equal-to (≥) and less-than-or-equal-to (≤), respectively. This is because many keyboards do not have the loose inequality operators. Even for systems having the loose operators, they will not be recognized. However, if you prefer, you may enter ">=" (and "<=") in place of ">" (and "<").

Parentheses as indicators of a preferred order of precedence are not accepted. All operations are ordered from left to right.

Comments may be placed anywhere in a model. A comment is denoted by an exclamation mark. Anything following an exclamation mark on the current line will be considered a comment. For example:

```
MAX 10 STD + 15 DLX   ! Max profit
SUBJECT TO
! Here are our factory capacity constraints
! for Standard and Deluxe computers
       STD < 10
       DLX < 12
! Here is the constraint on labor availability
       STD + 2 DLX < 16
END
```

LSreadLINDOFile() allows you to input comments, but they will not be stored with the model. The call to *LSreadLINDOFile*() does not store these comment. Therefore, if *LSwriteLINDOFile*() is called later, an equivalent model will be written, but the comments will be removed.

Constraints and the objective function may be split over multiple lines or combined on single lines. You may split a line anywhere except in the middle of a variable name or a coefficient. The following would be mathematically equivalent to our example (although not quite as easy to read):

```
MAX
       10
       STD  + 15 DLX  SUBJECT TO
STD
<
10
dlx < 12  STD + 2
dlx < 16 end
```

However, if the objective function appeared as follows:

```
MAX 10 ST
D + 1
5 DLX
SUBJECT TO
```

then *LSreadLINDOFile*() would return an error because the variable *STD* is split between lines and the coefficient 15 is also.

Only constant values—not variables—are permitted on the right-hand side of a constraint equation. Thus, an entry such as:

```
X > Y
```

would be rejected. Such an entry could be written as:

```
X - Y > 0
```

Conversely, only variables and their coefficients are permitted on the left-hand side of constraints. For instance, the constraint:

```
3X + 4Y - 10 = 0
```

is not permitted because of the constant term of -10 on the left-hand side. The constraint may be recast as:

```
3X + 4Y = 10
```

By default, all variables have lower bounds of zero and upper bounds of infinity.

Optional Modeling Statements

In addition to the three required model components of an objective function, variables, and constraints, a number of other optional modeling statements may appear in a model following the END statement. These statements and their functions appear in the table below:

Model Statement	**Function**
FREE <*Variable*>	Removes all bounds on <*Variable*>, allowing <*Variable*> to take on any real value, positive or negative.
GIN <*Variable*>	Makes <*Variable*> a general integer (i.e., restricts it to the set of nonnegative integers).
INT <*Variable*>	Makes <*Variable*> binary (i.e., restricts it to be either 0 or 1).
SLB <*Variable*> <*Value*>	Places a simple lower bound on <*Variable*> of <*Value*>. Use in place of constraints of form X = r.
SUB <*Variable*> <*Value*>	Places a simple upper bound on <*Variable*> of <*Value*>. Use in place of constraints of form X = r.
TITLE <*Title*>	Makes <*Title*> the title of the model.

Next, we will briefly illustrate the use of each of these statements.

FREE Statement

The default lower bound for a variable is 0. In other words, unless you specify otherwise, variables are not allowed to be negative. The FREE statement allows you to remove all bounds on a variable, so it may take on any real value, positive or negative.

The following small example illustrates the use of the FREE statement:

```
MIN 5X + Y
ST
  X+Y>5
  X-Y>7
END
FREE Y
```

Had we not set *Y* to be a free variable in this example, the optimal solution of *X* = 6 and *Y* = -1 would not have been found. Instead, given the default lower bound of 0 on *Y*, the solution *X* = 7 and *Y* = 0 would be returned.

GIN Statement

By default, all variables are assumed to be continuous. In other words, unless told otherwise, variables are assumed to be any nonnegative fractional number. In many applications, fractional values may be of little use (e.g., 2.5 employees). In these instances, you will want to make use of the general integer statement, GIN. The GIN statement followed by a variable name restricts the value of the variable to the nonnegative integers (0,1,2,…).

The following small example illustrates the use of the GIN statement:

```
MAX 11X + 10Y
ST
2X + Y < 12
X - 3Y > 1
END
GIN X
GIN Y
```

Had we not specified *X* and *Y* to be general integers in this model, the optimal solution of *X* = 6 and *Y* = 0 would not have been found. Instead, *X* and *Y* would have been treated as continuous and returned the solution of *X* = 5.29 and *Y* = 1.43.

Note also that simply rounding the continuous solution to the nearest integer values does not yield the optimal solution in this example. In general, rounded continuous solutions may be nonoptimal and, at worst, infeasible. Based on this, one can imagine that it can be very time consuming to obtain the optimal solution to a model with many integer variables. In general, this is true, and you are best off utilizing the GIN feature only when absolutely necessary.

INT Statement

Using the INT statement restricts a variable to being either 0 or 1. These variables are often referred to as *binary variables*. In many applications, binary variables can be very useful in modeling all-or-nothing situations. Examples might include such things as taking on a fixed cost, building a new plant, or buying a minimum level of some resource to receive a quantity discount.

The following small example illustrates the use of the INT statement:

```
MAX -100X + 20A + 12B
ST
  A - 10X < 0
  A + B < 11
  B < 7
END
INT X     !Make X 0/1
```

Had we not specified *X* to be binary in this example, a solution of *X* = .4, *A* = 4, and *B* = 7 for an objective value of 124 would not have been returned. Forcing *X* to be binary, you might guess that the optimal solution would be for *X* to be 0 because .4 is closer to 0 than it is to 1. If we round *X* to 0 and optimize for *A* and *B*, we get an objective of 84. In reality, a considerably better solution is obtained at *X* = 1, *A* = 10, and *B* = 1 for an objective of 112.

In general, rounded continuous solutions may be nonoptimal and, at worst, infeasible. Based on this, one can imagine that it can be very time consuming to obtain the optimal solution to a model with many binary variables. In general, this is true and you are best off utilizing the INT feature only when absolutely necessary.

SUB and SLB Statements

If you do not specify otherwise, LINDO API assumes variables are continuous (bounded below by zero and unbounded from above). That is, variables can be any positive fractional number increasing indefinitely. In many applications, this assumption may not be realistic. Suppose your facilities limit the quantity produced of an item. In this case, the variable that represents the quantity produced is bounded from above. Or, suppose you want to allow for backordering in a system. An easy way to model this is to allow an inventory variable to go negative. In which case, you would like to circumvent the default lower bound of zero. The *SUB* and *SLB* statements are used to alter the bounds on a variable. *SLB* stands for Simple Lower Bound and is used to set lower bounds. Similarly, *SUB* stands for Simple Upper Bound and is used to set upper bounds.

The following small example illustrates the use of the SUB and SLB:

```
MAX 20X + 30Y
ST
  X + 2Y < 120
END
SLB X 20
SUB X 50
SLB Y 40
SUB Y 70
```

In this example, we could have just as easily used constraints to represent the bounds. Specifically, we could have entered our small model as follows:

```
max 20x + 30y
st
   x + 2y < 120
   x > 20
   x < 50
   y > 40
   y < 70
end
```

This formulation would yield the same results, but there are two points to keep in mind. First, SUBs and SLBs are handled implicitly by the solver, and, therefore, are more efficient from a performance point of view than constraints. Secondly, SUBs and SLBs do not count against the constraint limit, allowing you to solve larger models within that limit.

TITLE Statement

This statement is used to associate a title with a model. The title may be any alphanumeric string of up to 74 characters in length. Unlike all the other statements that must appear after the END statement, the TITLE statement may appear before the objective or after the END statement of a model.

Here is an example of a small model with a title:

```
TITLE Your Title Here
MAX 20X + 30Y
ST
  X < 50
  Y < 60
  X + 2Y < 120
END
```

Appendix D: MPI File Format

The MPI (math program instructions) file format is a low level format for describing arbitrary nonlinear mathematical models. Expression of all relationships (linear or nonlinear) follows the same rules of instruction-list style interface described in Chapter 7, *Solving Nonlinear Programs*. The following example illustrates this:

```
* minimize= 2 * x0 + x1
* s.t.    -16 * x0       * x1   + 1 <= 0
*         - 4 * x0^2 - 4 * x1^2 + 1 <= 0
*          0 <= x0 <= 1
*                      0 <= x1 <= 1
BEGINMODEL  LSNLP1
VARIABLES
    X0  0.5  0.0  1.0  C
    X1  0.5  0.0  1.0  C
OBJECTIVES
  LSNLP1 LS_MIN
    EP_PUSH_NUM  2.0
    EP_PUSH_VAR  X0
    EP_MULTIPLY
    EP_PUSH_VAR  X1
    EP_PLUS
CONSTRAINTS
  R001 L
    EP_PUSH_NUM  -16.0
    EP_PUSH_VAR    X0
    EP_MULTIPLY
    EP_PUSH_VAR    X1
    EP_MULTIPLY
    EP_PUSH_NUM    1.0
    EP_PLUS
  R002 L
    EP_PUSH_NUM  -4.0
    EP_PUSH_VAR   X0
    EP_PUSH_NUM   2.0
    EP_POWER
    EP_MULTIPLY
    EP_PUSH_NUM  -4.0
    EP_PUSH_VAR   X1
    EP_PUSH_NUM   2.0
    EP_POWER
    EP_MULTIPLY
    EP_PLUS
    EP_PUSH_NUM   1.0
    EP_PLUS
ENDMODEL
```

Observe that an MPI file has the following structure:

3. Comment lines start with an "*" (asterisk),
4. There is a VARIABLES section that lists one line for each variable:
5. Its name, lower bound, an initial value, its upper bound, and its type, C(ontinuous) or I(nteger). A variable name must start with one of A-Z. Remaining characters must be one of A-Z, 0-9. Case does not matter (e.g., X1 is the same as x1). Names may have up to 255 characters.
6. There is an OBJECTIVES section that lists the name for the objective row and its type, LS_MIN or LS_MAX. This section also lists the instructions to compute the objective in postfix or Reverse Polish notation.
7. There is a CONSTRAINTS section that lists the name of each constraint and its type, L, G, E, or N for less-than-or-equal-to, greater-than-or-equal-to, equal-to, or not-constrained, respectively. This section also lists the instructions to compute the constraint in postfix or reverse Polish notation. Name conventions for constraints and objectives are the same as for variable names.

The instructions specify the operations to be performed on a LIFO(Last In First Out) stack of numbers. The instructions are of three main types:

8. Put(PUSH) a number on to the top of the stack,
9. Put(PUSH) the current value of a variable on to the top of the stack,
10. Perform some arithmetic operation on the top *k* elements of the stack and replace these *k* numbers with the result.

Refer to Chapter 7, *Solving Nonlinear Programs*, for more information on supported operators and functions.

Appendix E: SMPS File Format

The SMPS (stochastic mathematical programming standard) file format is an extension of the MPS format, described in Appendix B, for representing multistage linear programs. This format requires three files to completely define a stochastic multistage model.

CORE File

This is a standard MPS file to specify the deterministic version of the model, which is also called the *base model*. This file serves as the blueprint of the underlying model's nonzero structure when imposing stage information and stochasticity. This file generally has the extension '.mps'. Refer to Appendix B for details on MPS format.

TIME File

This file specifies the stage of each variable and constraint in the base model. The format of this file is similar to the MPS file where the information is provided in sections.

- TIME Specifies the name of the problem.
- PERIODS Specifies the stages in ascending order.
- ROWS Specifies the time stages of constraints.
- COLUMNS Specifies the time stages of variables.
- ENDATA Marks the end of staging data.

We call the base model (core-file), to be in temporal order if the variables and constraints are ordered with respect to their stage indices. Depending on whether the base model is in temporal order, time file can provide stage information implicitly or explicitly. The time-file usually has the extension '.time'.

Explict

If the core model is not in temporal order, the stage information should be given in an extended format. In PERIODS section, stage names should be given in ascending order of their indices. The keyword EMPLICIT is required in the second field of the PERIOD header. The stage information for variables and constraints are given in COLUMNS and ROWS sections, respectively. The following is the time-file associated with the Newsvendor model's in Chapter 8.

```
*0000000011111111112222222222333333333344444444445555555555
*>>4>678901<34>678901<34>6789012345<789>123456<89>123456789
TIME          NEWSVENDOR
PERIODS       EXPLICIT
    TIME1
    TIME2
    TIME3
COLUMNS
    X         TIME1
    I         TIME2
    L         TIME2
    S         TIME2
    Y         TIME3
    E         TIME3
    Z         TIME3
ROWS
    ROW1      TIME1
    ROW2      TIME2
    ROW3      TIME2
    ROW4      TIME2
    ROW5      TIME3
    PROFIT    TIME3
ENDATA
```

Implict

If the core model is in temporal order, then the stage information can be given in a compact way by simply specifying the first variable and constraint in each stage, where stage names are specified in ascending order of their indices. Optionally, the keyword IMPLICIT can be placed in the second field of the PERIOD header. The following is the time file associated with the Newsvendor model's in Chapter 8.

```
*0000000011111111112222222222333333333344444444445555555555
*>>4>678901<34>678901<34>6789012345<789>123456<89>123456789
TIME          NEWSVENDOR
PERIODS       IMPLICIT
    X         ROW1       TIME1
    I         ROW2       TIME2
    Y         ROW5       TIME3
```

STOCH File

This file identifies the stochastic elements in the base model, represented with the core-file, and the characteristics of their randomness (e.g. distribution type, distribution parameters, etc.). The format of this file is similar to the MPS file where the information is provided in sections.

- STOCH Specifies the name of the problem.
- INDEP Specifies the stage and univariate distribution of each independent random parameter.
- BLOCK Specifies the stage and joint distribution of random parameters.
- SCENARIOS Specifies an explicit scenario by identifying its parent scenario, how and when it differs from its parent and the stage at which it branched from its parent.
- ENDATA Marks the end of stochastic data.

Independent Distributions:

Independent distribution are identified with INDEP section, with the second field in the header being a keyword representing the distribution type, which can either be a parametric or a finite discrete distribution.

In the parametric case, such as the Normal distribution, the second field in INDEP header has to have the keyword NORMAL. Inside the INDEP section, the distribution of the parameters is represented as follows:

```
*000000001111111111222222222233333333334444444445555555555
*>>4>678901<34>678901<34>6789012345<789>123456<89>123456789
STOCH          NEWSVENDOR2
INDEP          NORMAL
*
    RHS1       ROW2            45.00000   TIME2      10
*
    Y          PROFIT          -3.00000   TIME3      2
ENDATA
```

In this example, the right-hand-side value in constraint [ROW2] takes a random value which is normally distributed with parameters μ=45, and σ=10. Similarly, variable [Y] in constraint [PROFIT] takes a random value which is normally distributed with parameters μ=-3, and σ=2.

In the finite discrete case, the second field of INDEP header should have the keyword DISCRETE. Inside the INDEP section, outcomes of each random parameter should be listed explicitly, where the sum of outcome probabilities should sum up to 1.0.

```
*000000001111111111222222222233333333334444444445555555555
*>>4>678901<34>678901<34>6789012345<789>123456<89>123456789
STOCH         NEWSVENDOR2
INDEP         DISCRETE
*
    RHS1      ROW2           90.00000   TIME2     0.4
    RHS1      ROW2           60.00000   TIME2     0.3
    RHS1      ROW2           30.00000   TIME2     0.3
*
    Y         PROFIT          9.00000   TIME3     0.3
    Y         PROFIT        -15.00000   TIME3     0.7
ENDATA
```

In this example, the right-hand-side value in constraint [ROW2] takes a random value from {90,60,30} with probabilities {0.4,0.3,0.3}. Similarly, variable [Y] in constraint [PROFIT] takes a random value from {9,-15} with probabilities {0.3,0.7}.

Joint Distributions with Intrastage Dependence:

Dependent distributions are identified with BLOCK sections, where each block corresponds to a vector of random parameters taking specified values jointly with a specified probabilitiy. The dependence is implict in the sense of joint distributions. The subsection BL within each BLOCK section marks each event (with its probability) listing the outcomes for a vector of random parameters.

```
*0000000011111111112222222222333333333344444444445555555555
*>>4>678901<34>678901<34>6789012345<789>123456<89>123456789
STOCH          NEWSVENDOR
BLOCKS         DISCRETE
 BL BLK0       TIME2      0.1200000000
    RHS1       ROW2       90.000000000
    Y          PROFIT     9.0000000000
 BL BLK0       TIME2      0.2800000000
    RHS1       ROW2       90.000000000
    Y          PROFIT     -15.00000000
 BL BLK0       TIME2      0.1500000000
    RHS1       ROW2       60.000000000
    Y          PROFIT     9.0000000000
 BL BLK0       TIME2      0.1500000000
    RHS1       ROW2       60.000000000
    Y          PROFIT     -15.00000000
 BL BLK0       TIME2      0.2700000000
    RHS1       ROW2       30.000000000
    Y          PROFIT     9.0000000000
 BL BLK0       TIME2      0.0300000000
    RHS1       ROW2       30.000000000
    Y          PROFIT     -15.00000000
ENDATA
```

In this example, the block called BLK0 lists the outcomes of the right-hand-side of constraints [ROW2] and [PROFIT]. Possible values are { (90,9), (90,-15), (60,9), (60,-15), (30,9), (30,-15)} with probabilities {0.12,0.28,0.15,0.15,0.27,0.03}.

Scenarios - Joint Distributions with Intertage Dependence:

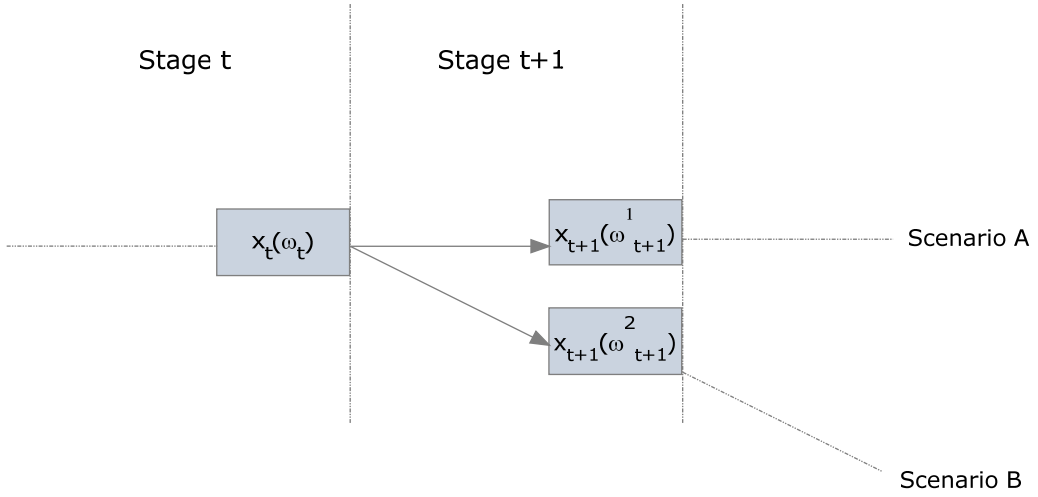
For models where discrete random parameters that belong to different stages are dependent, it is not possible to use the BLOCK structure to specify joint outcomes. This is because BLOCK structure requires the dependent random parameters to belong to the same stage. In such cases, it is required to input the stochastic data by specifying all scenarios explicitly with SCENARIOS section. For discrete distributions, this is the most general form for inputting a multistage SP because SCENARIOS section casts the entire scenario tree, irrespective of the type of dependence among randoms.

It could be a tedious task to enumerate all scenarios, therefore it is necessary to use a programming language or a script to generate scenarios programmatically writing them to a file in SCENARIOS format.

In a scenario tree, like the one given in Chapter 8, a scenario corresponds to a path from the root of the tree to one of the leaves. For each scenario, there is a one-to-one correspondence between each node on the path and a stage. One could think of a node as the point in time where decisions that belong to a stage are taken following the random outcomes that occur in that stage. The branches that emanate from a node represents the events associated with the next stage. Consequently, the set of all paths that branch from a node in a stage represents the future outcomes of all random parameters beyond that stage, namely the future as seen with respect to that node.

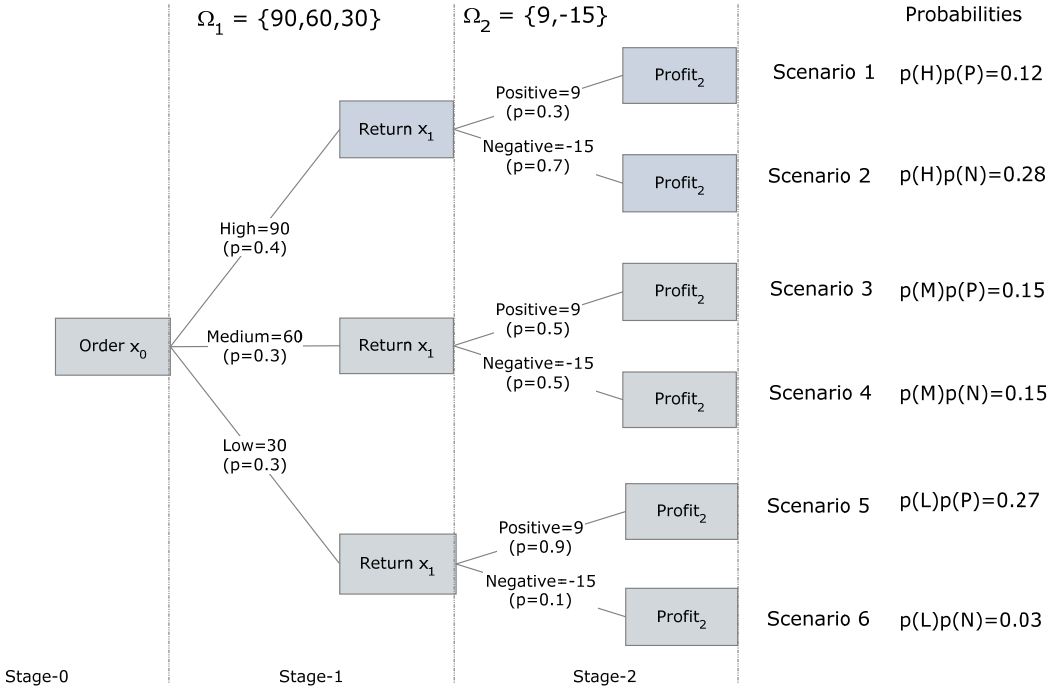
Given two scenarios A and B where they share the same path from the root up to (and including) stage *t*, we call

- The stage *t+1* to be the "branching stage" of scenario B from A,
- The scenario A to be the parent of scenario B.
- The outcomes of all random parameters up to (and including) stage *t* to be the same for both scenarios



The SCENARIOS section lists scenarios in a compact form, specifying how and when it differs from its parent scenario. The SC keyword marks the beginning of a scenario, which is followed by the name of the scenario, its parent's name and its probability . The probability of the scenario is to be computed by multiplying the conditional probabilities of all the nodes that resides on the path defining the scenario. The conditional probability of a node is the probability that the end-node occurs given the initial-node has occurred.

Consider the example from case 4 in the Newsvendor problem in Chapter 8, whose scenario tree is given as



This scenario tree can be represented in the following format using SCENARIOS section.

```
*0000000011111111112222222222333333333344444444445555555555
*>>4>678901<34>678901<34>6789012345<789>123456<89>123456789
STOCH          NEWSVENDOR
SCENARIOS      DISCRETE
 SC SCEN01     ROOT      0.1200000000   TIME1
    RHS1       ROW2      90.000000000
    Y          PROFIT    9.0000000000
 SC SCEN02     SCEN01    0.2800000000   TIME3
    Y          PROFIT    -15.00000000
 SC SCEN03     SCEN01    0.1500000000   TIME2
    RHS1       ROW2      60.000000000
 SC SCEN04     SCEN03    0.1500000000   TIME3
    Y          PROFIT    -15.00000000
 SC SCEN05     SCEN01    0.2700000000   TIME2
    RHS1       ROW2      30.000000000
 SC SCEN06     SCEN05    0.0300000000   TIME3
    Y          PROFIT    -15.00000000
ENDATA
```

In this example, the scenario tree is initiated by the base scenario called SCEN01, which lists inside its SC block a particular realization of all random parameters, namely the right-hand-side values for constraints [ROW2] and [PROFIT] to take values 90 and 9 respectively. The probability of the scenario is given as 0.12 and branching stage to be TIME1 (stage index 0). The parent of the base scenario is by default designated as the ROOT. This implies that this scenario has no parents and that it is the first scenario in the tree. The second scenario is SCEN02 and its parent is SCEN01, which was specified in the previous step. The scenario SCEN02 has a probability of 0.28 and branches of its parent SCEN01 at stage TIME2 (stage index 1). Inside its SC block, it only gives the random parameter and its value which makes SCEN02 different from SCEN1. This is the compact form described earlier, i.e. specifying a scenario's outcomes only by how and when it differs from its parent scenario. Continuing in this fashion, we achieve the complete representation of the scenario tree.

Appendix F: SMPI File Format

The SMPI (stochastic mathematical programming instructions) file format is an extension of the SMPS format, described in Appendix E, for representing multistage stochastic programs using an instruction list. While the SMPS format can only express linear and integer models, SMPI format can express all types of models including quadratic and general nonlinear stochastic models.

At the heart of the SMPI format lies the MPI format, which

- represents the core model using general mathematical expressions, and
- allows all random parameters in the SP to be referred with symbolically with EP_PUSH_SPAR macro.

The following illustrates a typical core-file for an SP model. Its only difference from a deterministic MPI file is in the use of EP_PUSH_SPAR macro, which marks each stochastic parameter in the model and allows them to be part of general mathematical expressions just like regular numeric constants or decision variables.

```
BEGINMODEL   newsboy_v5
! Number of Objective Functions:         1
! Number of Constraints        :         6
! Number of Variables          :         7
VARIABLES
!Name         Lower Bound       Initial Point         Upper Bound
Type
X 0  1.2345678806304932  1e+030        C
I 0  1.2345678806304932  1e+030        C
L 0  1.2345678806304932  1e+030        C
S 0  1.2345678806304932  1e+030        C
Y 0  1.2345678806304932  1e+030        C
E 0  1.2345678806304932  1e+030        C
Z 0  1.2345678806304932  1e+030        C
OBJECTIVES
  OBJ00000    MAXIMIZE
    EP_PUSH_VAR  Z
CONSTRAINTS
  ROW1    G
    EP_PUSH_VAR  X
    EP_PUSH_NUM       1
    EP_MINUS
  ROW2    E
    EP_PUSH_VAR  X
    EP_PUSH_VAR  I
    EP_MINUS
    EP_PUSH_VAR  L
    EP_PLUS
    EP_PUSH_SPAR     D
```

```
      EP_MINUS
    ROW3    E
      EP_PUSH_VAR  X
      EP_PUSH_VAR  I
      EP_MINUS
      EP_PUSH_VAR  S
      EP_MINUS
      EP_PUSH_NUM       0
      EP_MINUS
    ROW4    G
      EP_PUSH_VAR  X
      EP_PUSH_VAR  S
      EP_MINUS
      EP_PUSH_NUM       0
      EP_MINUS
    ROW5    E
      EP_PUSH_VAR  Y
      EP_PUSH_VAR  I
      EP_MINUS
      EP_PUSH_VAR  E
      EP_PLUS
      EP_PUSH_NUM       0
      EP_MINUS
    PROFIT    E
      EP_PUSH_NUM       60
      EP_PUSH_VAR  S
      EP_MULTIPLY
      EP_PUSH_NUM       30
      EP_PUSH_VAR  X
      EP_MULTIPLY
      EP_MINUS
      EP_PUSH_NUM       10
      EP_PUSH_VAR  I
      EP_MULTIPLY
      EP_MINUS
      EP_PUSH_NUM       5
      EP_PUSH_VAR  L
      EP_MULTIPLY
      EP_MINUS
      EP_PUSH_VAR  Y
      EP_PUSH_SPAR      R
      EP_MULTIPLY
      EP_PLUS
      EP_PUSH_NUM       10
      EP_PUSH_VAR  E
      EP_MULTIPLY
      EP_MINUS
      EP_PUSH_VAR  Z
      EP_MINUS
      EP_PUSH_NUM       0
      EP_MINUS
ENDMODEL
```

Like with SMPS format, the user has to define the time structure of the model with a TIME file. The TIME file in SMPI format uses an additional section, identified with keyword SVARS or SPARS, where time structure of random parameters are explicitly specified. The time structure of constraints and variables should also be specified explicitly. Implict specification is currently not supported in SMPI format.

```
*000000001111111111222222222233333333334444444445555555555
66
*>>4>678901<34>678901<34>6789012345<789>123456<89>1234567890<
TIME          NEWSVENDOR
PERIODS       EXPLICIT
    TIME1
    TIME2
    TIME3
COLUMNS
    X         TIME1
    I         TIME2
    L         TIME2
    S         TIME2
    Y         TIME3
    E         TIME3
    Z         TIME3
ROWS
    ROW1      TIME1
    ROW2      TIME2
    ROW3      TIME2
    ROW4      TIME2
    ROW5      TIME3
    PROFIT    TIME3
SPARS
    D         TIME2       63
    R         TIME3       9
ENDATA
```

Each random parameter that was referred in the Core-file should be listed in the TIME file along with their stage memberships and optionally a default value as the third field.

Finally, the user needs a STOCH file to specify the stochastic information for the SP model. In SMPS format, the random parameters was expressed by their location in the core model. In SMPI format, each random parameter has a unique name (a.k.a. an internal index), which can be used to refer each when specifying the information associated with it. Consequently, the STOCH file, whose format was laid out when explaining the SMPS format, can suitably be extended to support the indices of random parameters when expressing stochastic information using INDEP, BLOCK and SCENARIO sections. The keyword INST is used in field 1 of the line identifying the random parameter about which information is to be given.

A typical INDEP section in a STOCH file in SMPI format will be in the following

```
*000000001111111111222222222233333333334444444445555555555
*>>4>678901<34>678901<34>6789012345<789>123456<89>123456789
STOCHI         NEWSVENDOR
INDEP          NORMAL
*
    INST       D              45.00000   TIME2     10
*
    INST       R              -3.00000   TIME3     2

ENDATA
```

Similarly, the SCENARIOS section will be in the following form

```
*000000001111111111222222222233333333334444444445555555555
*>>4>678901<34>678901<34>6789012345<789>123456<89>123456789
STOCHI         NEWSVENDOR
SCENARIOS      DISCRETE
 SC SCEN01     ROOT       0.1200000000   TIME1
    INST       D          90.000000000
    INST       R          9.0000000000
 SC SCEN02     SCEN01     0.2800000000   TIME3
    INST       R          -15.00000000
 SC SCEN03     SCEN01     0.1500000000   TIME2
    INST       D          60.000000000
 SC SCEN04     SCEN03     0.1500000000   TIME3
    INST       R          -15.00000000
 SC SCEN05     SCEN01     0.2700000000   TIME2
    INST       D          30.000000000
 SC SCEN06     SCEN05     0.0300000000   TIME3
    INST       R          -15.00000000
ENDATA
```

As it can be seen from sample STOCH files, INST keyword identifies the keyword in the next field to be a random element and the stochastic information is provided in the same way as in SMPS format.

References

Birge, J. and F. Louveaux(1997), *Introduction to Stochastic Programming*, Springer.

L'Ecuyer, P., R. Simard, E. Chen, and W. Kelton(2002), "An Object-Oriented Random-Number Package with Many Long Streams and Substreams", *Operations Research*, vol. 50, no. 6, pp. 1073-1075.

INDEX

1

100% rule, 434

A

absolute optimality tolerance, 73
absolute value, 339, 350, 368
Add Module command, 269, 430
adding
  constraints, 194, 518
  variables, 194, 195, 520
addition, 339
AddressOf operator, 233, 235, 429
advanced routines, 221, 541
algebraic reformulation, 86
algorithm
  barrier, 295
  generalized reduced gradient, 3
alternate optima, 158
analysis routines, 210, 534
analyzing models and solutions, 433
AND function, 339, 343
angular block structure, 445, 534
annuity, 340
antithetic variate, 98, 249, 407
antithetic variates, 416
API
  callback functions, 227
  error messages, 565
  examples, 259
  function definitions, 17
arc sine, 340
arc tangent, 340
arguments, right-hand side, 452
ASCII text format, 26
asset investing, 304
asymmetric *Q* matrix, 300
automatic differentiation, 349
auxiliary routines, 552
available parameters, 53, 99
average, 342

B

backward transformation, 221, 541
barrier algorithm, 295
barrier solver, 77, 94
  iterations, 227
  license, 92
  solving, 53, 58, 60, 80, 139, 265
basis, 542
  crossover, 58
  cuts, 227
  forward transformation, 222
  getting, 144, 147, 491
  loading, 486
  MIPs, 493
  parameters, 73
  warm start, 135
Beasley, J., 304
Beta distribution, 416
Big M, 70, 351
binary variables, 26, 167, 504, 587, 588
binomial distribution, 341
Binomial distribution, 416
Birge, J., 411
black-box interface, 330, 353
  example, 355
blanks, 573, 574
block structured models, 444
  finding, 210, 534
  getting, 213, 537
  loading, 137, 488
  parameters, 55
bounds
  assets invested, 304
  best bounds, 212, 536
  defaults, 286, 293, 587, 588, 589
  free variables, 573, 587
  global optimization, 87
  MATLAB, 475, 476, 503, 520, 552, 553
  modifying, 206, 209, 529, 533
  MPS files, 573
  name, 119, 168, 477, 505
  objective bounds, 167, 432, 504
  ranges, 214, 433, 538
  real bound, 78
  risk of loss, 300
  running time, 76
  SUB/SLB, 587, 589
  type, 575
  variable upper/lower, 118, 166, 196, 276, 587
branch-and-bound
  cuts, 70
  limits, 69, 79, 82, 83, 89
  solver, 142, 287, 293, 375, 491
  solver status, 230, 547

branching
branch count, 227, 432
branch direction, 73, 75, 85, 86
global optimization, 82
priorities, 71, 124, 136, 138, 486, 489
strong branching, 78
variable branching, 78, 486
BTRAN, 221, 541
building an application, 267

C

C example, 259, 421, 424
C++ example, 280
debugging, 435–44
callback functions, 355, 421
definitions, 227, 546
double precision, 423
examples, 421–32
frequency, 53
MIPs, 235, 279, 550
query routines, 227, 546
callback management routines, 227, 546
callback.bas, 429
CALLBACKTYPE, 422
calling conventions, 452, 558
capitalization, 6, 567, 574
cardinality constraints, 304
Cauchy distribution, 416
cdecl protocol, 422
CheckErr(), 273
Chisquare distribution, 416
Chi-squared distribution, 341
Cholesky decomposition, 326
class module, 429
ClassWizard, 282
clique cuts, 227
coefficients
adding, 194, 196, 520
backward transformation, 221, 541
C++ example, 264, 285, 286
coefficient matrix, 10, 166, 265
forward transformation, 542
getting, 164, 165, 167, 502, 503, 504, 552
left-hand side, 587
linearization, 351
loading, 117, 118, 473, 476, 553
modifying, 204, 527
number of, 164, 167, 265, 502, 504
quadratic, 121, 197
range of, 56
reduction, 61, 69, 76, 81
right-hand side, 275, 518
sparse format, 114
storing, 276
Visual Basic example, 291, 292
column
column length, 12, 165, 196, 285, 293
column start, 11, 12, 285, 291
file format, 195, 196, 520
MATLAB, 556
names, 119, 477
nonlinear, 120, 170, 478, 506, 507, 508, 509
comments, 586
compiling, 266
complement function, 339
complementarity, 352
cone optimization, 311
conjunction, 339
constant term, 115, 117, 165, 205, 206, 335, 474, 476, 503, 530, 552, 553
constraints, 291, 585
adding, 194, 518
C++ example, 264, 286
cardinality, 304
complementarity, 352
cuts, 72
deleting, 200, 524
equal to, 116, 474
errors, 565
forcing, 351
get, 160, 162, 164, 165, 498, 500, 502, 503, 552
greater than, 116, 152, 474
GUB, 69
index of, 127, 164, 223, 224, 543, 544
internal index, 501
left-hand sides, 587
less than, 116, 152, 474
limit, 589
loading, 118, 476, 553
matrix, 118, 166, 265, 476, 503
modifying, 204, 205, 207, 527, 528
names, 119, 168, 501, 585
nonlinear data, 169, 506, 508, 509
number of, 91, 115, 117, 152, 167, 473, 474, 476, 503, 504
Pluto Dogs example, 285
quadratic, 121, 174, 197, 295
ranges, 215, 433, 538
right-hand sides, 264, 531, 586
selective evaluation, 60
splitting, 586
status, 144
storing, 276
violated, 55, 63
Visual Basic example, 292
continuous model, 139, 152, 490
continuous variables, 139, 148, 158, 352, 589
priorities, 136
contra cuts, 227

converting models to SOCP form, 321
convex models, 54, 64, 350, 353, 354, 371
convexification, 83
core file, 34, 36
core model, 131, 132, 133, 155, 157, 187, 191, 393, 411
correlation, 415
cosine, 340
covariance, 295
crashing, 60, 64
creating
  environment, 453
  model, 453
creation routines, 21
crossover, 58, 140
cutoff value, 54, 58, 72, 77, 81
cuts
  depth, 70
  frequency, 70
  max passes, 71
  total generated, 227
  types of, 69, 70, 72

D

data
  fields, 282
  formulation, 164, 165, 167, 502, 503, 504
  getting, 162, 421, 500
  global, 21, 421, 428, 431
  lines, 575
  loading, 124
  name, 119, 168, 477, 505
  passing, 276
  quadratic, 510, 511
  storing, 21
  structures, 21, 262, 266, 428
  types, 17, 42, 263
debug, 435
  example, 438
  parameters, 93
decision variables, 285, 291
decomposition, viii, 445
  angular structures, 447
  Dantzig-Wolfe, 556
  finding, 211, 534
  getting, 488, 537
  loading, 138
  parameters, 55, 86
  total, 446
default bounds, 286, 293, 587, 588, 589
definitions, 17
deletion routines, 21, 200, 203, 524
  examples, 266, 277
  MATLAB, 454
  nonlinear programming, 330
  variables, 526
Delphi, 277
delta tolerance, 70, 83, 351
derivatives, 335
  accuracy, 349
  calculating, 224, 226, 336, 544, 545
  discontinuous, 337
  examples, 355
  finite differences, 60, 63
  getting, 170, 171, 507
  setting, 234, 549
deterministic equivalent, 3, 20, 37, 97
Devex pricing, 57, 58
differentiation, 349
dimensions of model, 42, 275
direction
  of constraints, 205
  of objective, 264, 285, 291
  to branch, 73, 75
disaggregation, 69, 227
discontinuous derivatives, 337
discrete variables, 352
disjunction, 339
Distribution Function Macros, 256
division, 339
double precision, 348, 368
  callback functions, 423, 432
  getting parameters, 44, 46, 466, 468
  parameters, 42
  setting parameters, 48, 50, 469, 471
dual
  models, 29, 30, 54, 459
  objective, 227
  reductions, 61, 76, 81
  simplex, 57, 60, 69, 77, 80, 94, 139, 265
  solution, 148
  values, 145, 423, 492, 493
  writing, 459
dual angular structure, 210, 445, 534

E

e, 340
ector Push, 345
educational license, 92
eigenvalue, 295
embedded blanks, 573, 574
END, 585
engineering design, 311
enumeration solver, 74
environment, 22
  creating, 21, 259, 273, 453
  deleting, 21, 22, 454
  space, 21

variables, 267
EP_ABS, 339
EP_ACOS, 340
EP_ACOSH, 346
EP_AND, 339
EP_ASIN, 340
EP_ASINH, 345
EP_ATAN, 340
EP_ATAN2, 340
EP_ATANH, 346
EP_AVG, 342
EP_COS, 340
EP_COSH, 345
EP_DIVIDE, 339
EP_EQUAL, 339
EP_EXP, 340
EP_EXPOINV, 343
EP_EXT_AND, 343
EP_FALSE, 340
EP_FLOOR, 340
EP_FPA, 340
EP_FPL, 341
EP_GTHAN, 339
EP_GTOREQ, 339
EP_IF, 340
EP_LGM, 340
EP_LN, 339
EP_LNX, 346
EP_LOG, 339
EP_LOGB, 346
EP_LOGX, 346
EP_LTHAN, 339
EP_LTOREQ, 339
EP_MAX, 343
EP_MIN, 342
EP_MINUS, 339
EP_MOD, 340
EP_MULTINV, 344
EP_MULTIPLY, 339
EP_NEGATE, 339
EP_NO_OP, 339
EP_NORMDENS, 343
EP_NORMINV, 343
EP_NORMSINV, 346
EP_NOT, 339
EP_NOT_EQUAL, 339
EP_NPV, 343
EP_OR, 339
EP_PBN, 341
EP_PCX, 341
EP_PEB, 341
EP_PEL, 341
EP_PERCENT, 339
EP_PFD, 342
EP_PFS, 342
EP_PHG, 342
EP_PI, 339
EP_PLUS, 339
EP_POWER, 339
EP_PPL, 341
EP_PPS, 341
EP_PSL, 340
EP_PSN, 340
EP_PTD, 341
EP_PUSH_NUM, 343
EP_PUSH_OR, 343
EP_PUSH_VAR, 343
EP_RAND, 342
EP_SIGN, 340
EP_SIN, 340
EP_SINH, 345
EP_SQR, 345
EP_SQRT, 339
EP_SUM, 342
EP_SUMIF, 344
EP_SUMPROD, 344
EP_TAN, 340
EP_TANH, 345
EP_TRIAINV, 343
EP_TRUE, 340
EP_TRUNC, 346
EP_UNIFINV, 344
EP_USER, 342
EP_USER operator, 346
EP_USRCOD, 344
EP_VLOOKUP, 344
EP_VMULT, 345
EP_VPUSH_NUM, 345
EP_VPUSH_VAR, 345
EP_WRAP, 341
equal to
constraints, 118, 164, 165, 194, 502
error messages, 565
operators, 339, 586
quadratic programs, 295
Erlang loss, 341
error codes, 219, 220, 464, 565
error handling routines, 53, 219, 273, 464
EVMU, 109, 417
EVPI, 97, 109, 417
examples
callback functions, 421–32
debugging, 438
linear programs, 259
MATLAB, 554
programming in C, 259, 421
Visual Basic, 287
exclamation mark, 586
exp( ) function, 340
expiration, 58, 79, 91, 568

exponential distribution, 343, 416

F

F distribution, 342, 416
false, 340
feasibility tolerance, 55, 63
fields, 282
file formats, 25
  ASCII text format, 26
  column format, 195, 196, 520
  error messages, 565
  LINDO, 585
  LINGO, 25, 32, 461
  MPI, 27, 375, 457, 565, 591, 593, 601
  MPS, 25, 26, 169, 573
  row format, 194, 585
file input, 7
finance, 300
finite differences, 335
  black-box interface, 355
  coefficients, 121
  derivatives, 60, 63, 234, 549
  gradients, 235, 336
  instruction-list interface, 349
finite source queue, 342
first order approximations, 59
fixed variables, 72, 81, 573
floating point tolerance, 82
FLOOR function, 340
flow cover, 69, 227
forcing constraints, 351
form module, 429
formatted MPS file, 26
formulation data, 164, 165, 167, 502, 503, 504
forward transformation, 222, 542
four vector representation, 12
FREE, 587
free variables, 54, 573, 587, 588
frequency of callbacks, 53
frequency of cuts, 70
frontend, 329
FTRAN, 222, 542
full rank, 295
Funcalc(), 334
functions
  definitions, 17
  objective, 57, 291, 585, 586
  postfix notation, 339
  prefixes, 17
  prototypes, 262
functions to callback, 355, 421
  definitions, 227, 546
  frequency, 53
  MIPs, 235, 279, 550

G

Gamma distribution, 416
gamma function, 340
gaussian distributions, 353
GCD cuts, 69, 227
general integers, 167, 279, 504, 587, 588
generalized upper bound, 69
Geometric distribution, 416
getting
  constraints, 162, 164, 500, 502
  data, 162, 421, 500
  parameters, 43, 44, 127, 466, 467, 468
  variable types, 516
GIN, 167, 279, 504, 587, 588
global data, 21, 421, 428, 431
global optimality, 62
global optimization
  cuts, 227
  non-convex models, 353
  nonlinear models, 350, 354
  parameters, 82, 91
  quadratic programs, 295
  solving, 141, 490
global solver, vii, 3, 92, 354, 375
Gomory cuts, 69, 227
Gradcalc(), 336
gradient, 3, 63, 120, 329, 336, 371, 478
greater than, 118, 164, 165, 194, 502
  constraints, 152
  errors, 565
  example, 264, 285, 291
  operator, 586
  postfix notation, 339
grey-box interface, 330, 346
  example, 381
GUB cuts, 69, 227
Gumbel distribution, 416

H

handler code, 282
hashing, 238
header file, 21, 53, 262, 263, 268, 452
heuristic, 72, 74
histogram, 191
Hungarian notation, 17, 452
Hypergeometric distribution, 416
hypergeometric probability, 342

I

IF( ) function, 340
IIS, 31, 435, 460
  finding, 211

getting, 216, 539
MATLAB, 535
parameters, 93
incumbent solution, 83, 84, 227, 430
indefinite, 295
independent block structure, 444
index
of a row, 166, 176, 196, 276, 504, 513, 520
of constraints, 127, 164, 223, 224, 543, 544
inequality operators, 586
infeasibilities, 227
MATLAB, 535, 539
primal infeasibility, 423, 546
rounded solutions, 588, 589
solver status, 433
infeasible solution, 31, 216, 435, 460
infinity, 587
infix notation, 337
inheriting, 42
initial values, 135, 139, 179, 486, 487, 489, 515
initialization of solver, 135, 486
inner product, 344
Input/Output, of models, 25
instruction-list interface, 115, 116, 330, 337, 474
example, 361
instructition format, 348
INT, 587, 588
integer part, 340
integer programming. *See also* mixed-integer programming
callback functions, 235, 430, 432, 550
constraint cuts, 72
cut level, 69, 70
examples, 279
getting, 44, 46
heuristics, 74
internal index, 178, 501, 514
loading, 124
optimality tolerance, 73
setting, 49, 50
slack values, 150, 152, 497
integer variables
binary, 587, 588
block structure, 137
bounded, 573
branching priorities, 136, 138, 489
general, 167, 279, 504, 587, 588
integer feasible tolerance, 73, 77
limit, 91
parameters, 42
solving for, 139, 148, 158
variable status, 144, 147, 213
integrality, 69, 287, 293
interface, 329, 421
black-box, 330, 353, 355
callback function, 428
grey-box, 346, 381
instruction list, 330, 337, 361
java, viii
MATLAB, viii, 449
nonlinear, 329
interior point algorithm, 295
interior point solver, 58, 60, 77, 80, 92, 94, 139, 265
Interior-Point Solver Programs
parameters, 65
internal error, 566
internal index
constraints, 501
getting, 164, 167
variables, 178, 179, 514, 515
interrupt solver, 53, 422, 430, 568
inverse of standard Normal, 343, 346
inverse transform of cdf, 416
investing, 304
irreducibly inconsistent set, 31, 435, 460
finding, 211
getting, 216, 539
MATLAB, 535
parameters, 93
irreducibly unbounded set, 31, 437, 460
finding, 211
getting, 217, 540
MATLAB, 535
parameters, 93
iterations, 227
barrier, 432
callback functions, 428, 430
iteration limit, 56, 63, 65, 567
nonlinear, 432
simplex, 432
IUS, 31, 435, 437, 460
finding, 211
getting, 217, 540
MATLAB, 535
parameters, 93

J

Jacobian, 169, 170, 172, 349, 478, 506, 507, 509
java interface, viii
JNI, viii

K

K-best solutions, 158
Kendall tau, 415
knapsack cuts, 69, 227
knapsack solver, 74

L

Laplace distribution, 416
Latin hypercube sampling, 98, 249, 407, 408, 409, 416
Latin square sampling, 98, 249, 407, 408, 409, 416
lattice cuts, 69, 227
leading blanks, 573
left-hand sides, 587
  arguments, 452
length of column, 12, 196, 285, 293
length of objective, 369
less than, 118, 164, 165, 194, 502
  constraints, 152
  errors, 565
  example, 264, 275
  operator, 586
  postfix notation, 339
license
  barrier, 92, 265, 297, 315
  C++ example, 6
  educational, 92
  error messages, 567, 568
  expiration, 91
  global, 92
  license key, 21, 23
  MATLAB, 453, 455
  nonlinear, 92, 297, 315
  reading, 24
  runtime, 92
  trial, 91
license key, 6
LIFO stack, 592
lifting cuts, 69
limits
  branch-and-bound, 79, 82
  constraints, 589
  integer variables, 91
  iteration, 56, 63, 65, 567
  license expiration, 91
  time limit, 58, 70, 79, 80, 89, 91, 568
  variables, 91
LINDO contact information, ix
LINDO format, 25, 585
  reading, 25, 456
  writing, 29, 32, 458, 461
lindo.bas, 269
lindo.h, 262, 268, 269, 431
lindo.par, 8
linear loss function, 340, 341
linear models, 353
linear programming, 1, 78, 259
  getting data, 503
  loading, 476
linear solver, 2
linearity, 61, 62, 337, 351
linearization, viii, 3, 70, 350, 369
LINGO format, 25, 32
  writing, 461
linking, 266
Linux, 8
LMBinPack.m, 556
LMreadf.m, 555
lndapi40.lic, 6, 24
loading
  models, 117, 476
  variables, 482, 483, 484, 485
Loading Core Model, 394
Loading the Stochastic Structure, 399
Loading the Time Structure, 397
locally optimal, 352, 371
location, 422
logarithm, 335, 339
Logarithmic distribution, 416
logical operators, 350
Logistic distribution, 416
Lognormal distribution, 416
long variable, 269
looping, 286, 292
loose inequality operators, 586
Louveaux, F., 411
lower bounds
  adding, 196, 520
  best, 212
  getting, 166, 475, 503, 552
  LINDO files, 587
  loading, 118, 476, 553
  MIPs, 71
  modifying, 206, 529
  MPS files, 573
  nonlinear programs, 117, 336
  objective, 167, 504
  SLB, 587, 589
  Visual Basic example, 276
LS_BASTYPE_ATLO, 18, 19
LS_BASTYPE_ATUP, 18, 19
LS_BASTYPE_BAS, 18, 19
LS_BASTYPE_FNUL, 18, 19
LS_BASTYPE_SBAS, 18, 19
LS_CONETYPE_QUAD, 19
LS_CONETYPE_RQUAD, 19
LS_CONTYPE_EQ, 19
LS_CONTYPE_FR, 19
LS_CONTYPE_GE, 19
LS_CONTYPE_LE, 19
LS_DERIV_BACKWARD_DIFFERENCE, 63
LS_DERIV_CENTER_DIFFERENCE, 63
LS_DERIV_FORWARD_DIFFERENCE, 63
LS_DERIV_FREE, 63
LS_DINFO_MIP_OBJ, 432

LS_DINFO_MIP_SOLOBJVAL_LST_BRANCH, 432
LS_DINFO_MIPBESTBOUND, 432
LS_DINFO_SAMP_KURTOSIS, 112
LS_DINFO_SAMP_MEAN, 112
LS_DINFO_SAMP_SKEWNESS, 112
LS_DINFO_SAMP_STD, 112
LS_DINFO_STOC_ABSOPT_GAP, 109
LS_DINFO_STOC_DINFEAS, 109
LS_DINFO_STOC_EVOBJ, 109
LS_DINFO_STOC_NUM_COLS_DETEQE, 111
LS_DINFO_STOC_NUM_COLS_DETEQI, 111
LS_DINFO_STOC_NUM_NODES, 110
LS_DINFO_STOC_NUM_NODES_STAGE, 110
LS_DINFO_STOC_NUM_ROWS_DETEQE, 111
LS_DINFO_STOC_NUM_ROWS_DETEQI, 111
LS_DINFO_STOC_NUM_SCENARIOS, 110
LS_DINFO_STOC_PINFEAS, 109
LS_DINFO_STOC_RELOPT_GAP, 109
LS_DINFO_STOC_TOTAL_TIME, 110
LS_DPARAM_CALLBACKFREQ, 53, 422
LS_DPARAM_GOP_BNDLIM, 83, 87
LS_DPARAM_GOP_BOXTOL, 82
LS_DPARAM_GOP_DELTATOL, 83
LS_DPARAM_GOP_FLTTOL, 82
LS_DPARAM_GOP_OPTTOL, 82, 84
LS_DPARAM_GOP_WIDTOL, 83, 84
LS_DPARAM_IPM_BASIS_REL_TOL_S, 66
LS_DPARAM_IPM_BASIS_TOL_S, 66
LS_DPARAM_IPM_BASIS_TOL_X, 66
LS_DPARAM_IPM_BI_LU_TOL_REL_PIV, 66
LS_DPARAM_IPM_TOL_DSAFE, 66
LS_DPARAM_IPM_TOL_INFEAS, 65
LS_DPARAM_IPM_TOL_MU_RED, 66
LS_DPARAM_IPM_TOL_PATH, 65
LS_DPARAM_IPM_TOL_PFEAS, 65
LS_DPARAM_MIP_ABSOPTTOL, 73
LS_DPARAM_MIP_ADDCUTOBJTOL, 72
LS_DPARAM_MIP_ADDCUTPER, 72
LS_DPARAM_MIP_ADDCUTPER_TREE, 72
LS_DPARAM_MIP_AOPTTIMLIM, 72
LS_DPARAM_MIP_BIGM_FOR_INTTOL, 67
LS_DPARAM_MIP_CUTOFFOBJ, 76
LS_DPARAM_MIP_CUTOFFVAL, 77
LS_DPARAM_MIP_CUTTIMLIM, 70
LS_DPARAM_MIP_DELTA, 70, 351
LS_DPARAM_MIP_FP_TIMLIM, 69
LS_DPARAM_MIP_FP_WEIGTH, 68
LS_DPARAM_MIP_HEUMINTIMLIM, 72, 74
LS_DPARAM_MIP_INTTOL, 73
LS_DPARAM_MIP_LBIGM, 70, 351
LS_DPARAM_MIP_LSOLTIMLIM, 80
LS_DPARAM_MIP_MINABSOBJSTEP, 80
LS_DPARAM_MIP_PEROPTTOL, 72, 73
LS_DPARAM_MIP_PSEUDOCOST_WEIGT, 81
LS_DPARAM_MIP_REDCOSTFIX_CUTOFF, 72
LS_DPARAM_MIP_REDCOSTFIX_CUTOFF_TREE, 81
LS_DPARAM_MIP_RELINTTOL, 73, 77
LS_DPARAM_MIP_RELOPTTOL, 73
LS_DPARAM_MIP_TIMLIM, 79
LS_DPARAM_NLP_FEASTOL, 63
LS_DPARAM_NLP_PSTEP_FINITEDIFF, 60
LS_DPARAM_NLP_REDGTOL, 63
LS_DPARAM_OBJPRINTMUL, 57
LS_DPARAM_SOLVER_CUTOFFVAL, 54, 58
LS_DPARAM_SOLVER_FEASTOL, 44, 55
LS_DPARAM_SOLVER_IUSOL, 56
LS_DPARAM_SOLVER_OPTTOL, 55
LS_DPARAM_STOC_ABSOPTTOL, 98
LS_DPARAM_STOC_RELOPTTOL, 97
LS_DPARAM_STOC_TIME_LIM, 97
LS_FORMATTED_MPS, 26, 457
LS_IINFO_DIST_TYPE, 111
LS_IINFO_ITER, 432
LS_IINFO_MIP_ACTIVENODES, 432
LS_IINFO_MIP_BRANCHCOUNT, 432
LS_IINFO_MIP_LPCOUNT, 432
LS_IINFO_MIP_LTYPE, 432
LS_IINFO_MIP_NEWIPSOL, 432
LS_IINFO_MIP_SOLSTATUS_LAST_BRANCH, 432
LS_IINFO_MIP_STATUS, 432
LS_IINFO_NUM_STOCPAR_AIJ, 110
LS_IINFO_NUM_STOCPAR_INSTR_CONS, 110
LS_IINFO_NUM_STOCPAR_INSTR_OBJS, 110
LS_IINFO_NUM_STOCPAR_LB, 110
LS_IINFO_NUM_STOCPAR_OBJ, 110
LS_IINFO_NUM_STOCPAR_RHS, 110
LS_IINFO_NUM_STOCPAR_UB, 110
LS_IINFO_SAMP_SIZE, 111
LS_IINFO_STOC_BAR_ITER, 109
LS_IINFO_STOC_NLP_ITER, 109
LS_IINFO_STOC_NUM_BENDERS_FCUTS, 111
LS_IINFO_STOC_NUM_BENDERS_OCUTS, 111
LS_IINFO_STOC_NUM_COLS_BEFORE_NODE, 111
LS_IINFO_STOC_NUM_COLS_CORE, 111
LS_IINFO_STOC_NUM_COLS_DETEQE, 111
LS_IINFO_STOC_NUM_COLS_DETEQI, 111
LS_IINFO_STOC_NUM_COLS_NAC, 111
LS_IINFO_STOC_NUM_COLS_STAGE, 111
LS_IINFO_STOC_NUM_NODE_MODELS, 110
LS_IINFO_STOC_NUM_NODES, 110
LS_IINFO_STOC_NUM_NODES_STAGE, 110
LS_IINFO_STOC_NUM_QCP_CONS_DETEQE, 112
LS_IINFO_STOC_NUM_ROWS_BEFORE_NODE, 111

LS_IINFO_STOC_NUM_ROWS_CORE, 111
LS_IINFO_STOC_NUM_ROWS_DETEQE, 111
LS_IINFO_STOC_NUM_ROWS_DETEQI, 111
LS_IINFO_STOC_NUM_ROWS_NAC, 111
LS_IINFO_STOC_NUM_ROWS_STAGE, 111
LS_IINFO_STOC_NUM_SCENARIOS, 110
LS_IINFO_STOC_NUM_STAGES, 110
LS_IINFO_STOC_STAGE_BY_NODE, 110
LS_IINFO_STOC_STATUS, 110
LS_IIS_ADD_FILTER, 20, 436
LS_IIS_DEFAULT, 20, 436
LS_IIS_DEL_FILTER, 20, 436
LS_IIS_DFBS_FILTER, 20, 436
LS_IIS_ELS_FILTER, 20, 437
LS_IIS_FSC_FILTER, 20, 436
LS_IIS_GBS_FILTER, 20, 436
LS_IIS_NORM_FREE, 20, 437
LS_IIS_NORM_INFINITY, 20, 437
LS_IIS_NORM_ONE, 20, 437
LS_IMAT_AIJ, 20
LS_INFINITY, 18, 118, 166, 196
LS_IPARAM_ MIP_USECUTOFFOBJ, 77
LS_IPARAM_ALLOW_CNTRLBREAK, 53
LS_IPARAM_BARRIER_PROB_TO_SOLVE, 54
LS_IPARAM_BARRIER_SOLVER, 53
LS_IPARAM_CHECK_FOR_ERRORS, 53
LS_IPARAM_DECOMPOSITION_TYPE, 55
LS_IPARAM_GOP_ALGREFORMMD, 86
LS_IPARAM_GOP_BBSRCHMD, 86
LS_IPARAM_GOP_BRANCH_LIMIT, 88
LS_IPARAM_GOP_BRANCHMD, 85
LS_IPARAM_GOP_CORELEVEL, 89
LS_IPARAM_GOP_DECOMPPTMD, 86
LS_IPARAM_GOP_HEU_MODE, 89
LS_IPARAM_GOP_LPSOPT, 89
LS_IPARAM_GOP_LSOLBRANLIM, 89
LS_IPARAM_GOP_MAXWIDMD, 83, 84
LS_IPARAM_GOP_OPT_MODE, 87
LS_IPARAM_GOP_OPTCHKMD, 84
LS_IPARAM_GOP_POSTLEVEL, 85
LS_IPARAM_GOP_PRELEVEL, 85
LS_IPARAM_GOP_PRINTLEVEL, 86
LS_IPARAM_GOP_RELBRNDMD, 87
LS_IPARAM_GOP_SUBOUT_MODE, 89
LS_IPARAM_GOP_TIMLIM, 83
LS_IPARAM_GOP_USE_NLPSOLVE, 89
LS_IPARAM_GOP_USEBNDLIM, 87
LS_IPARAM_IIS_ANALYZE_LEVEL, 93
LS_IPARAM_IIS_INFEAS_NORM, 94
LS_IPARAM_IIS_ITER_LIMIT, 94
LS_IPARAM_IIS_METHOD, 93
LS_IPARAM_IIS_PRINT_LEVEL, 94
LS_IPARAM_IIS_REOPT, 94
LS_IPARAM_IIS_TIME_LIMIT, 95
LS_IPARAM_IIS_USE_EFILTER, 93
LS_IPARAM_IIS_USE_GOP, 93
LS_IPARAM_IIS_USE_SFILTER, 94
LS_IPARAM_INSTRUCT_LOADTYPE, 53
LS_IPARAM_IPM_MAX_ITERATIONS, 66
LS_IPARAM_IPM_NUM_THREADS, 67
LS_IPARAM_IPM_OFF_COL_TRH, 66
LS_IPARAM_IUS_ANALYZE_LEVEL, 93
LS_IPARAM_IUS_TOPOPT, 94
LS_IPARAM_LIC_BARRIER, 92
LS_IPARAM_LIC_CONSTRAINTS, 91
LS_IPARAM_LIC_DAYSTOEXP, 91
LS_IPARAM_LIC_DAYSTOTRIALEXP, 91
LS_IPARAM_LIC_EDUCATIONAL, 92
LS_IPARAM_LIC_GLOBAL, 92
LS_IPARAM_LIC_GOP_INTEGERS, 91
LS_IPARAM_LIC_GOP_NONLINEARVARS, 91
LS_IPARAM_LIC_INTEGERS, 91
LS_IPARAM_LIC_NONLINEAR, 92
LS_IPARAM_LIC_NONLINEARVARS, 91
LS_IPARAM_LIC_NUMUSERS, 92
LS_IPARAM_LIC_PLATFORM, 91
LS_IPARAM_LIC_RUNTIME, 92
LS_IPARAM_LIC_VARIABLES, 91
LS_IPARAM_LP_PRELEVEL, 59
LS_IPARAM_LP_PRINTLEVEL, 57, 62
LS_IPARAM_LP_SCALE, 56
LS_IPARAM_MAXCUTPASS_TREE, 71
LS_IPARAM_MIP_AGGCUTLIM_TOP, 80
LS_IPARAM_MIP_AGGCUTLIM_TREE, 80
LS_IPARAM_MIP_ANODES_SWITCH_DF, 78
LS_IPARAM_MIP_BRANCH_LIMIT, 79
LS_IPARAM_MIP_BRANCH_PRIO, 71
LS_IPARAM_MIP_BRANCHDIR, 73
LS_IPARAM_MIP_BRANCHRULE, 75
LS_IPARAM_MIP_CUTDEPTH, 70
LS_IPARAM_MIP_CUTFREQ, 70
LS_IPARAM_MIP_CUTLEVEL_TOP, 69
LS_IPARAM_MIP_CUTLEVEL_TREE, 70
LS_IPARAM_MIP_DUAL_SOLUTION, 80
LS_IPARAM_MIP_FP_ITRLIM, 69
LS_IPARAM_MIP_FP_MODE, 68
LS_IPARAM_MIP_FP_OPT_METHOD, 69
LS_IPARAM_MIP_HEU_MODE, 68
LS_IPARAM_MIP_HEULEVEL, 72, 74
LS_IPARAM_MIP_ITRLIM, 79
LS_IPARAM_MIP_KEEPINMEM, 73
LS_IPARAM_MIP_MAKECUT_INACTIVE_CO UNT, 65, 68
LS_IPARAM_MIP_MAXCUTPASS_TOP, 71
LS_IPARAM_MIP_MAXNONIMP_CUTPASS, 71
LS_IPARAM_MIP_NODESELRULE, 75
LS_IPARAM_MIP_PRE_ELIM_FILL, 65, 68
LS_IPARAM_MIP_PRELEVEL, 76
LS_IPARAM_MIP_PRELEVEL_TREE, 81

LS_IPARAM_MIP_PREPRINTLEVEL, 76
LS_IPARAM_MIP_PRINTLEVEL, 76
LS_IPARAM_MIP_PSEUDOCOST_RULE, 81
LS_IPARAM_MIP_REOPT, 77
LS_IPARAM_MIP_SCALING_BOUND, 71
LS_IPARAM_MIP_SOLVERTYPE, 74
LS_IPARAM_MIP_STRONGBRANCHDONUM, 68
LS_IPARAM_MIP_STRONGBRANCHLEVEL, 78
LS_IPARAM_MIP_SWITCHFAC_SIM_IPM, 78
LS_IPARAM_MIP_TOPOPT, 80
LS_IPARAM_MIP_TREEREORDERLEVEL, 78
LS_IPARAM_MIP_USE_CUTS_HEU, 67
LS_IPARAM_MIP_USE_ENUM_HEU, 81
LS_IPARAM_MIP_USE_INT_ZERO_TOL, 67
LS_IPARAM_MPS_OBJ_WRITESTYLE, 54
LS_IPARAM_NLP_AUTODERIV, 61, 349
LS_IPARAM_NLP_AUTOHESS, 64
LS_IPARAM_NLP_CONVEX, 64
LS_IPARAM_NLP_CONVEXRELAX, 63
LS_IPARAM_NLP_CR_ALG_REFORM, 63
LS_IPARAM_NLP_DERIV_TYPE, 63
LS_IPARAM_NLP_FEASCHK, 62
LS_IPARAM_NLP_ITRLMT, 63, 65
LS_IPARAM_NLP_LINEARITY, 62, 351
LS_IPARAM_NLP_LINEARZ, 61, 351
LS_IPARAM_NLP_MAXLOCALSEARCH, 64, 375
LS_IPARAM_NLP_PRELEVEL, 61
LS_IPARAM_NLP_QUADCHK, 64
LS_IPARAM_NLP_SOLVE_AS_LP, 59
LS_IPARAM_NLP_SOLVER, 59
LS_IPARAM_NLP_STALL_ITRLMT, 64
LS_IPARAM_NLP_STARTPOINT, 63
LS_IPARAM_NLP_SUBSOLVER, 60
LS_IPARAM_NLP_USE_CRASH, 60
LS_IPARAM_NLP_USE_LINDO_CRASH, 64
LS_IPARAM_NLP_USE_SELCONEVAL, 60
LS_IPARAM_NLP_USE_SLP, 60
LS_IPARAM_NLP_USE_STEEPEDGE, 60
LS_IPARAM_OBJSENSE, 57
LS_IPARAM_PROB_TO_SOLVE, 58
LS_IPARAM_SOL_REPORT_STYLE, 53
LS_IPARAM_SOLVER_IPMSOL, 58, 140
LS_IPARAM_SOLVER_PRE_ELIM_FILL, 59
LS_IPARAM_SOLVER_RESTART, 57
LS_IPARAM_SOLVER_TIMLMT, 58
LS_IPARAM_SOLVER_USE_CONCURRENT_OPT, 67
LS_IPARAM_SOLVER_USECUTOFFVAL, 58
LS_IPARAM_SPLEX_DPRICING, 57
LS_IPARAM_SPLEX_DUAL_PHASE, 59
LS_IPARAM_SPLEX_ITRLMT, 56
LS_IPARAM_SPLEX_PPRICING, 57
LS_IPARAM_SPLEX_REFACFRQ, 53
LS_IPARAM_STOC_BUCKET_SIZE, 97
LS_IPARAM_STOC_CALC_EVPI, 97
LS_IPARAM_STOC_DEBUG_LEVEL, 97
LS_IPARAM_STOC_DETEQ_TYPE, 97
LS_IPARAM_STOC_ITER_LIM, 96
LS_IPARAM_STOC_MAX_NUMSCENS, 97
LS_IPARAM_STOC_METHOD, 96
LS_IPARAM_STOC_NODELP_PRELEVEL, 97
LS_IPARAM_STOC_PRINT_LEVEL, 96
LS_IPARAM_STOC_REOPT, 96
LS_IPARAM_STOC_RG_SEED, 96
LS_IPARAM_STOC_SAMP_CONT_ONLY, 97
LS_IPARAM_STOC_SAMPSIZE_NODE, 96
LS_IPARAM_STOC_SAMPSIZE_SPAR, 96
LS_IPARAM_STOC_SHARE_BEGSTAGE, 97
LS_IPARAM_STOC_TOPOPT, 96
LS_IPARAM_TIMLMT, 58
LS_IPARAM_VER_NUMBER, 58
LS_IROW_OBJ, 20
LS_IROW_VFX, 20
LS_IROW_VLB, 20
LS_IROW_VUB, 20
LS_JCOL_INST, 20
LS_JCOL_RHS, 20
LS_JCOL_RLB, 20
LS_JCOL_RUB, 20
LS_LINK_BLOCKS_BOTH, 55
LS_LINK_BLOCKS_COLS, 55
LS_LINK_BLOCKS_FREE, 55
LS_LINK_BLOCKS_NONE, 55
LS_LINK_BLOCKS_ROWS, 55
LS_MAX, 18, 57, 117, 165, 276
LS_MAX_ERROR_MESSAGE_LENGTH, 219
LS_METHOD_BARRIER, 19, 60, 139
LS_METHOD_DSIMPLEX, 19, 60, 139
LS_METHOD_FREE, 19, 139
LS_METHOD_NLP, 19, 139
LS_METHOD_PSIMPLEX, 19, 60, 139, 265
LS_METHOD_STOC_ALD, 20
LS_METHOD_STOC_DETEQ, 20
LS_METHOD_STOC_FREE, 20
LS_METHOD_STOC_NBD, 20
LS_MIN, 18, 57, 115, 117, 165, 264, 285, 291, 474
LS_MIP_SET_CARD, 20
LS_MIP_SET_SOS1, 19
LS_MIP_SET_SOS2, 19
LS_MIP_SET_SOS3, 19
LS_NMETHOD_CONOPT, 59
LS_NMETHOD_FREE, 59
LS_NMETHOD_MSW_GRG, 59
LS_PROB_SOLVE_DUAL, 54
LS_PROB_SOLVE_FREE, 54
LS_PROB_SOLVE_PRIMAL, 54
LS_SOLUTION_MIP, 19

LS_SOLUTION_MIP_OLD, 19
LS_SOLUTION_OPT, 19
LS_SOLUTION_OPT_IPM, 19
LS_SOLUTION_OPT_OLD, 19
LS_STATUS_CUTOFF, 18
LS_STATUS_INFORUNB, 18
LS_STATUS_LOADED, 18
LS_STATUS_LOCAL_INFEASIBLE, 18
LS_STATUS_LOCAL_OPTIMAL, 18
LS_STATUS_NEAR_OPTIMAL, 18
LS_STATUS_NUMERICAL_ERROR, 18
LS_STATUS_UNKNOWN, 18
LS_STATUS_UNLOADED, 18
LS_UNFORMATTED_MPS, 26
LS_VARTYPE_BIN, 19
LS_VARTYPE_CONT, 19
LS_VARTYPE_INT, 19
LSaddCones(), 193, 517
LSaddConstraints(), 194, 195, 518, 520
LSaddContinuousIndep (), 131
LSaddDiscreteBlocks (), 132
LSaddDiscreteIndep (), 130
LSaddNLPAj(), 198
LSaddNLPobj(), 199
LSaddQCterms(), 197
LSaddScenario (), 133
LSaddSETS(), 195, 519, 521, 522, 523
LSaddVariables(), 194, 195, 520
LSbuildStringData(), 126
LScalcConFunc(), 223, 543
LScalcConGrad(), 224, 544
LScalcObjFunc(), 225, 543
LScalcObjGrad(), 226, 545
LScreateEnv(), 21, 263, 269, 330, 453, 567
  quadratic programming, 297, 316
LScreateModel(), 21, 22, 263, 269, 273, 330, 453
  quadratic programming, 297, 316
LScreateRG (), 241
LSdeleteAj(), 202
LSdeleteCones(), 199, 523
LSdeleteConstraints(), 200, 524
LSdeleteEnv(), 21, 22, 266, 277, 297, 316, 454
  nonlinearprogramming, 330
LSdeleteModel(), 22, 23, 454
LSdeleteNLPobj(), 201
LSdeleteQCterms(), 200, 524
LSdeleteSemiContVars(), 202, 525
LSdeleteSETS(), 203, 525, 526, 527
LSdeleteString(), 127
LSdeleteStringData(), 126
LSdeleteVariables(), 203, 526
LSdisposeRG (), 243
LSdoBTRAN(), 221, 541
LSdoFTRAN(), 222, 542
LSenv()
  creating, 21, 453
  deleting, 22, 454
  error messages, 219
  getting, 43, 44, 466, 467
  setting, 47, 48, 49, 469, 470
LSERR_ARRAY_OUT_OF_BOUNDS, 571
LSERR_BAD_CONSTRAINT_TYPE, 565
LSERR_BAD_DECOMPOSITION_TYPE, 565
LSERR_BAD_DISTRIBUTION_TYPE, 571
LSERR_BAD_LICENSE_FILE, 565
LSERR_BAD_MODEL, 565
LSERR_BAD_MPI_FILE, 565
LSERR_BAD_MPS_FILE, 565
LSERR_BAD_OBJECTIVE_SENSE, 565
LSERR_BAD_SMPI_CORE_FILE, 569
LSERR_BAD_SMPI_STOC_FILE, 569
LSERR_BAD_SMPS_CORE_FILE, 569
LSERR_BAD_SMPS_STOC_FILE, 569
LSERR_BAD_SMPS_TIME_FILE, 569
LSERR_BAD_SOLVER_TYPE, 565
LSERR_BAD_VARIABLE_TYPE, 565
LSERR_BASIS_BOUND_MISMATCH, 565
LSERR_BASIS_COL_STATUS, 565
LSERR_BASIS_INVALID, 565
LSERR_BASIS_ROW_STATUS, 565
LSERR_BLOCK_OF_BLOCK, 566
LSERR_BOUND_OUT_OF_RANGE, 566
LSERR_CANNOT_OPEN_CORE_FILE, 569
LSERR_CANNOT_OPEN_FILE, 566
LSERR_CANNOT_OPEN_STOC_FILE, 569
LSERR_CANNOT_OPEN_TIME_FILE, 569
LSERR_CHECKSUM, 566
LSERR_COL_BEGIN_INDEX, 566
LSERR_COL_INDEX_OUT_OF_RANGE, 566
LSERR_COL_NONZCOUNT, 566
LSERR_CORE_BAD_NUMSTAGES, 570
LSERR_CORE_BAD_STAGE_INDEX, 570
LSERR_CORE_BAD_STRUCTURE, 570
LSERR_CORE_INVALID_SPAR_INDEX, 569
LSERR_CORE_NOT_IN_TEMPORAL_ORDER, 571
LSERR_CORE_SPAR_COUNT_MISMATCH, 569
LSERR_CORE_SPAR_NOT_FOUND, 569
LSERR_CORE_SPAR_VALUE_NOT_FOUND, 570
LSERR_CORE_TIME_MISMATCH, 570
LSERR_DATA_TERM_EXIST, 568
LSERR_DIST_BAD_CORRELATION_TYPE, 572
LSERR_DIST_INVALID_NUMPARAM, 571
LSERR_DIST_INVALID_PARAMS, 571
LSERR_DIST_INVALID_PROBABILITY, 571
LSERR_DIST_INVALID_SD, 571
LSERR_DIST_INVALID_X, 571

LSERR_DIST_NO_DERIVATIVE, 571
LSERR_DIST_NO_PDF_LIMIT, 571
LSERR_DIST_ROOTER_ITERLIM, 571
LSERR_DIST_SCALE_OUT_OF_RANGE, 571
LSERR_DIST_SHAPE_OUT_OF_RANGE, 571
LSERR_DIST_TRUNCATED, 571
LSERR_EMPTY_COL_STAGE, 572
LSERR_EMPTY_ROW_STAGE, 572
LSERR_ERRMSG_FILE_NOT_FOUND, 566
LSERR_ERROR_IN_INPUT, 115, 118, 123, 474, 481, 566
LSERR_GOP_BRANCH_LIMIT, 566
LSERR_GOP_FUNC_NOT_SUPPORTED, 566
LSERR_ILLEGAL_NULL_POINTER, 566
LSERR_INDEX_DUPLICATE, 566
LSERR_INDEX_OUT_OF_RANGE, 118, 123, 481, 566
LSERR_INFO_NOT_AVAILABLE, 566
LSERR_INFO_UNAVAILABLE, 570
LSERR_INST_INVALID_BOUND, 568
LSERR_INST_MISS_ELEMENTS, 568
LSERR_INST_SYNTAX_ERROR, 568
LSERR_INST_TOO_SHORT, 568
LSERR_INSTRUCT_NOT_LOADED, 566
LSERR_INTERNAL_ERROR, 566
LSERR_INVALID_ERRORCODE, 567
LSERR_ITER_LIMIT, 567
LSERR_LAST_ERROR, 567, 568
LSERR_MIP_BRANCH_LIMIT, 567
LSERR_MISSING_TOKEN_NAME, 570
LSERR_MISSING_TOKEN_ROOT, 570
LSERR_MODEL_ALREADY_LOADED, 567
LSERR_MODEL_NOT_LINEAR, 567
LSERR_MODEL_NOT_LOADED, 567
LSERR_NO_ERROR, 567
LSERR_NO_LICENSE_FILE, 567
LSERR_NO_METHOD_LICENSE, 567
LSERR_NO_VALID_LICENSE, 567
LSERR_NOT_CONVEX, 567
LSERR_NOT_SORTED_ORDER, 568
LSERR_NOT_SUPPORTED, 567
LSERR_NUMERIC_INSTABILITY, 567
LSERR_OLD_LICENSE, 567
LSERR_OUT_OF_MEMORY, 567
LSERR_PARAMETER_OUT_OF_RANGE, 568
LSERR_RG_ALREADY_SET, 572
LSERR_RG_NOT_SET, 571
LSERR_RG_SEED_NOT_SET, 572
LSERR_ROW_INDEX_OUT_OF_RANGE, 568
LSERR_SCEN_INDEX_OUT_OF_SEQUENCE, 569
LSERR_STEP_TOO_SMALL, 568
LSERR_STOC_BAD_ALGORITHM, 570
LSERR_STOC_BAD_PRECISION, 570
LSERR_STOC_BLOCK_SAMPLING_NOT_SUPPORTED, 572
LSERR_STOC_EVENTS_NOT_LOADED, 572
LSERR_STOC_INVALID_CDF, 571
LSERR_STOC_INVALID_SAMPLE_SIZE, 571
LSERR_STOC_INVALID_SCENARIO_CDF, 569
LSERR_STOC_MISSING_BNDNAME, 570
LSERR_STOC_MISSING_OBJNAME, 570
LSERR_STOC_MISSING_PARAM_TOKEN, 571
LSERR_STOC_MISSING_RHSNAME, 570
LSERR_STOC_MISSING_RNGNAME, 570
LSERR_STOC_MODEL_ALREADY_PARSED, 569
LSERR_STOC_MODEL_NOT_LOADED, 569
LSERR_STOC_NODE_INFEASIBLE, 570
LSERR_STOC_NODE_UNBOUNDED, 570
LSERR_STOC_NOT_DISCRETE, 571
LSERR_STOC_NULL_EVENT_TREE, 570
LSERR_STOC_OUT_OF_SAMPLE_POINTS, 572
LSERR_STOC_SAMPLE_ALREADY_GENERATED, 572
LSERR_STOC_SAMPLE_ALREADY_LOADED, 572
LSERR_STOC_SAMPLE_NOT_GENERATED, 572
LSERR_STOC_SAMPLE_SIZE_TOO_SMALL, 572
LSERR_STOC_SCENARIO_LIMIT, 571
LSERR_STOC_SCENARIO_SAMPLING_NOT_SUPPORTED, 572
LSERR_STOC_SPAR_NOT_FOUND, 569
LSERR_STOC_TOO_MANY_SCENARIOS, 570
LSERR_STOC_TREE_ALREADY_INIT, 572
LSERR_TIME_BAD_NUMSTAGES, 570
LSERR_TIME_BAD_TEMPORAL_ORDER, 570
LSERR_TIME_LIMIT, 568
LSERR_TIME_NUMSTAGES_NOT_SET, 572
LSERR_TIME_SPAR_COUNT_MISMATCH, 569
LSERR_TIME_SPAR_NOT_EXPECTED, 569
LSERR_TIME_SPAR_NOT_FOUND, 569
LSERR_TOO_SMALL_LICENSE, 568
LSERR_TOTAL_NONZCOUNT, 568
LSERR_TRUNCATED_NAME_DATA, 568
LSERR_UNABLE_TO_SET_PARAM, 568
LSERR_USER_FUNCTION_NOT_FOUND, 568
LSERR_USER_INTERRUPT, 568
LSERR_VARIABLE_NOT_FOUND, 568
LSfindBlockStructure(), 210, 446, 534
LSfindIIS(), 211, 435, 535
LSfindIUS(), 211, 535
LSfreeGOPSolutionMemory(), 238
LSfreeHashMemory(), 238
LSfreeMIPSolutionMemory(), 239
LSfreeSolutionMemory(), 239
LSfreeSolverMemory(), 240

LSgetBasis(), 144, 491
LSgetBestBounds(), 212, 536
LSgetBlockStructure(), 213, 537
LSgetBoundRanges(), 214, 433, 435, 538
LSgetCallback, 279
LSgetCallbackInfo(), 227, 422, 428, 430, 431, 546
LSgetConeDatai(), 160, 498
LSgetConeIndex(), 161, 499
LSgetConeNamei(), 161, 499
LSgetConstraintDatai(), 162, 500
LSgetConstraintIndex(), 163, 501
LSgetConstraintNamei(), 163, 501
LSgetConstraintRanges(), 215, 433, 434, 538
LSgetDeteqModel (), 184
LSgetDiscreteBlockOutcomes, 188
LSgetDiscreteBlocks, 187
LSgetDiscreteIndep, 189
LSgetDistrRV (), 242
LSgetDoubleRV (), 241
LSgetDualMIPSolution(), 279
LSgetDualSolution(), 145, 492
  nonlinear programming, 330
  quadratic programming, 297, 316
LSgetEnvDouParameter(), 44, 466
LSgetEnvIntParameter(), 44, 467
LSgetEnvParameter(), 43, 466
LSgetErrorMessage(), 219, 273, 464
LSgetErrorRowIndex, 220
LSgetErrorRowIndex(), 464
LSgetFileError(), 220, 465
LSgetHistogram, 191
LSgetIIS(), 216, 539
LSgetInfo(), 146, 279, 492
  nonlinear programming, 330
  quadratic programming, 297, 316
LSgetInitSeed (), 242
LSgetInt32RV (), 242
LSgetIUS(), 217, 540
LSgetLPConstraintDatai(), 164, 502
LSgetLPData(), 165, 503
LSgetLPVariableDataj(), 167, 504
LSgetMIPBasis(), 147, 493
LSgetMIPCallbackInfo(), 230, 279, 431, 547
LSgetMIPDualSolution(), 148, 493
LSgetMIPPrimalSolution(), 148, 158, 494
LSgetMIPReducedCosts(), 149, 279, 494
LSgetMIPSlacks(), 150, 279, 495
LSgetMIPSolution(), 279
LSgetModelDouParameter(), 46, 467, 468
LSgetModelIntParameter(), 46, 351, 468
LSgetModelParameter(), 45, 51, 467
LSgetNameData(), 168, 505
LSgetNextBestMIPSoln (), 158
LSgetNLPConstraintDatai(), 169
LSgetNLPConstraintDatai(), 506
LSgetNLPData(), 170, 507
LSgetNLPObjectiveData(), 171
LSgetNLPObjectiveData(), 508
LSgetNLPVariableDataj(), 172
LSgetNLPVariableDataj(), 509
LSgetNodeDualSolution, 156, 157, 158
LSgetNodeDualSolution (), 156
LSgetNodeListByScenario (), 185
LSgetNodePrimalSolution, 154
LSgetNodePrimalSolution (), 154
LSgetNodeReducedCost (), 38
LSgetNodeSlacks, 156
LSgetNodeSlacks (), 156
LSgetObjective(), 265, 277
LSgetObjectiveRanges(), 218, 433, 434, 540
LSgetPrimalSolution(), 150, 495
  C++ example, 265
  MATLAB, 495
  nonlinear programming, 330
  quadratic programming, 297, 316
  Visual Basic example, 277
LSgetProbabilityByNode (), 184
LSgetProbabilityByScenario (), 183
LSgetQCData(), 173, 510
LSgetQCDatai(), 174, 511
LSgetReducedCosts(), 151, 496
LSgetReducedCostsCone(), 151, 496
LSgetSampleSizes, 190
LSgetScenarioDualSolution (), 157
LSgetScenarioIndex (), 183
LSgetScenarioModel, 192
LSgetScenarioName, 182
LSgetScenarioName (), 182
LSgetScenarioObjective, 154
LSgetScenarioObjective (), 154
LSgetScenarioPrimalSolution, 155
LSgetScenarioPrimalSolution (), 155
LSgetScenarioReducedCost (), 155
LSgetScenarioSlacks (), 158
LSgetSemiContData(), 175, 512
LSgetSETSData(), 176, 513
LSgetSETSDatai(), 177, 514
LSgetSlacks(), 152, 497
LSgetSolution(), 153, 497
LSgetStageIndex (), 181
LSgetStageName (), 180
LSgetStocParData (), 186
LSgetStocParIndex (), 181
LSgetStocParName (), 182
LSgetStocParOutcomes, 186
LSgetStocParOutcomes (), 185
LSgetStocParSample, 254
LSgetStringValue(), 127
LSgetVariableIndex(), 178, 514
LSgetVariableNamej(), 179, 515

LSgetVariableStages, 191
LSgetVarStartPoint(), 179, 515
LSgetVarType(), 180, 516
LSgetVersionInfo(), 23, 455
LSgetxxxyyyParameter(), 53
LSloadBasis(), 135, 486
LSloadBlockStructure(), 137, 139, 488
LSloadConeData (), 114
LSloadConeData(), 316, 473
LSloadConstraintStages (), 129
LSloadInstruct(), 115, 349, 369, 474
LSloadLicenseString(), 23, 455
LSloadLPData(), 117, 279
  C++ example, 287
  integer programming, 279
  MATLAB, 476
  nonlinear programming, 330
  quadratic programming, 297, 316
  Visual Basic example, 276, 292, 293
LSloadNameData(), 33, 119, 477
LSloadNLPData(), 120, 330, 478
LSloadQCData(), 121, 297, 299, 479
LSloadSampleSizes (), 128
LSloadSemiContData(), 122, 480
LSloadSETSData(), 123, 481
LSloadStocParData (), 130
LSloadStocParNames (), 134
LSloadString(), 125
LSloadStringData(), 125
LSloadVariableStages (), 129
LSloadVarPriorities(), 136, 486
LSloadVarStartPoint(), 136, 487
LSloadVarType(), 118, 124, 279, 287, 293, 482, 483, 484, 485
  integer programming, 279
  quadratic programming, 297, 316
LSmodel
  creating, 22, 453
  deleting, 23, 454
  getting, 467, 468
  loading, 117, 119, 124, 476, 477, 482, 483, 484, 485
  setting, 470, 471
LSmodifyAj(), 204, 527
LSmodifyCone(), 204, 528
LSmodifyConstraintType(), 205, 528
LSmodifyLowerBounds(), 206, 529
LSmodifyObjConstant(), 205, 206, 530
LSmodifyObjective(), 207, 530
LSmodifyRHS(), 207, 531
LSmodifySemiContVars(), 208, 531
LSmodifySET(), 208, 532
LSmodifyUpperBounds(), 209, 533
LSmodifyVariableType(), 209, 533
LSoptimize(), 139
  C++ example, 265
  MATLAB, 490
  nonlinear programming, 330
  quadratic programming, 297, 316
  Visual Basic example, 276
LSreadBasis(), 28
LSreadEnvParameter(), 51, 472
LSreadLINDOFile(), 25, 456, 585, 586
LSreadModelParameter(), 51, 472, 473
LSreadMPIFile(), 27, 457
LSreadMPSFile(), 26, 457, 573
LSreadSMPIFile(), 35
LSreadSMPSFile (), 34
LSreadVarPriorities(), 138, 489
LSreadVarStartPoint(), 139, 489
LSsampCreate (), 245
LSsampDelete (), 245
LSsampEvalDistr (), 248
LSsampGenerate (), 249
LSsampGetCIPoints (), 250
LSsampGetCIPointsPtr (), 251
LSsampGetCorrelationMatrix (), 251
LSsampGetDiscretePdfTable (), 246
LSsampGetDistrParam (), 247
LSsampGetInfo (), 253
LSsampGetPoints (), 249
LSsampGetPointsPtr (), 250
LSsampInduceCorrelation (), 252
LSsampLoadDiscretePdfTable (), 246
LSsampSetDistrParam (), 247
LSsampSetRG (), 248
LSsampSetUserDistr (), 247
LSsetCallback(), 227, 231, 279, 421, 422, 428
  MATLAB, 546, 547
  Visual Basic example, 429
LSsetDistrParamRG (), 244
LSsetDistrRG (), 244
LSsetEnvDouParameter(), 48, 422, 469
LSsetEnvIntParameter(), 49, 470
LSsetEnvLogFunc(), 232
LSsetEnvParameter(), 47, 469
LSsetFuncalc(), 233, 330, 334, 355, 548
LSsetGradcalc(), 234, 330, 336, 549
LSsetMIPCallback(), 235, 279, 430, 431
  MATLAB, 547, 550
LSsetModelDouParameter(), 50, 351, 471
LSsetModelIntParameter(), 50, 349, 351, 471
LSsetModelLogFunc(), 236, 551
LSsetModelParameter(), 49, 470
LSsetNumStages (), 128
LSsetRGSeed (), 243
LSsetUsercalc(), 237, 347, 551
LSsetxxxyyyParameter(), 53
LSsolveGOP(), 139, 141, 490
LSsolveMIP(), 139, 142, 279, 491

C++ example, 287
nonlinear programming, 370
quadratic programming, 297, 316
Visual Basic example, 293
LSsolveSP (), 143
LSstocInfo
LS_IINFO_STOC_SIM_ITER, 109
LSwriteBasis(), 28
LSwriteDeteqLINDOFile (), 37
LSwriteDeteqMPSFile (), 37
LSwriteDualLINDOFile(), 29, 458
LSwriteDualMPSFile(), 30, 459
LSwriteEnvParameter(), 52
LSwriteIIS(), 31, 460
LSwriteIUS(), 31, 460
LSwriteLINDOFile(), 32, 461, 586
LSwriteLINGOFile(), 32, 461
LSwriteModelParameter(), 52
LSwriteMPIFile(), 27
LSwriteMPSFile(), 33, 462, 573
LSwriteNodeSolutionFile (), 39
LSwriteScenarioLINDOFile (), 41
LSwriteScenarioMPIFile (), 40
LSwriteScenarioMPSFile (), 40
LSwriteScenarioSolutionFile (), 39
LSwriteSMPIFile(), 36
LSwriteSMPSFile (), 36
LSwriteSolution(), 34, 463
LSXgetLPData(), 552
LSXloadLPData(), 553
lump sum, 341

M

Macintosh, 8
macros, 262
_LINDO_DLL_, 268
APIERRORSETUP, 263
LS_DINFO_POBJ, 265
makefile.win, 267, 428
Markowitz model, 300
mathematical guarantee, 351
MATLAB, viii, 449
matrix, 10, 118, 166, 265, 286, 292, 476, 503
block structured, 137, 445, 488
covariance, 300
nonlinear, 121
quadratic, 121, 174, 197
sparse, 329
maximization, 57, 117, 165, 350, 476, 503, 552, 553
memory, 230, 239, 240, 423, 567
memory management routines, 238
MEX-file, 449
Microsoft Foundation Class, 280
minimization, 57, 117, 165, 350, 476, 503, 552, 553
minus, 586
mixed-integer programs, 148, 158
branch-and-bound, 142, 491
callback functions, 235, 279, 550
cut level, 69, 70
data loading, 124
example, 279
parameters, 67
query routines, 287, 294
solution, 494
mixed-integer solver, 2
mod function, 340, 341
model
analyzing, 433
block structured, 210, 213, 444
continuous, 139, 152, 490
convex, 350, 371
creating, 22, 259, 273, 453
data, 21
deleting, 22, 23, 454
dimensions, 42, 275
dual, 29, 30, 459
I/O routines, 25
loading routines, 114, 117, 473
modification routines, 193, 517
monitoring, 421, 431
nonlinear, 329
primal, 29
query routines, 160, 498
reading, 25
smooth, 350
writing, 25
model and solution analysis routines, 534
modification routines, 193, 517
modifying variable types, 533
modules, 429
modulo, 286, 292
Monte Carlo Sampling, 406
MPI format, 27, 375, 457, 565, 591, 593, 601
MPS format, 25, 573
debugging, 435–44
error messages, 565
extended, 296
LMreadf.m, 555
names, 169
reading, 26, 457
SOCP, 314
writing, 30, 33, 458, 462, 463
MS Windows, 8
multinomial distribution, 344
multiple choice, 577
multiplication, 339
Multistage Decision Making Under Uncertainty, 389

multistart solver, vii, 3, 8, 59, 64, 227, 350, 352, 353, 371, 423
mxLINDO, 449
routines, 452

N

names
column, 119, 477
constraints, 119, 168, 501, 585
data, 119, 168, 477, 505
getting, 168, 178, 179, 514, 515
hashing, 238
LINDO files, 585
loading, 119, 125
MATLAB, 514
MPS files, 573
row, 119, 477
natural logarithm, 339
necessary set, 436, 437
negation, 339, 349
negative semi-definite, 295
negative variables, 573, 587
Negativebinomial distribution, 416
New Project command, 287
newsvendor problem, 392, 394, 400
nmake, 266, 267, 428
node selection rule, 75, 86
non-convex models, 350, 354
nonlinear programs, vii, 62, 329
constraint data, 169, 506, 508, 509
getting data, 170, 507
iterations, 227
loading data, 120, 478
objective data, 171
optimization, 139
parameters, 59, 91, 92
variable data, 172
nonlinear solver, 3, 295
nonoptimal solutions, 588, 589
non-smooth models, viii, 350, 354
nonzero coefficients
adding, 194, 502, 518
C++ example, 286
coefficient matrix, 166, 286, 292
columns, 196, 285, 286, 291, 292, 520
getting, 164, 503
loading, 118, 476
number of, 164, 167, 265, 476, 502, 504
sparse format, 114, 473
storing, 276
variables, 167
vectors, 221, 541, 542
Visual Basic example, 292
norm minimization, 316
Normal cdf, 340
normal density, 343
Normal distribution, 416
not equal to, 339
notation
Hungarian, 17, 452
postfix, 337, 368, 592
Reverse Polish, 337, 592
NP-hard, 354
numeric error, 87, 567

O

object oriented, 280
objective
adding, 196, 520
bounds, 167, 432, 504
C++ example, 264, 285
constant value, 115, 117, 165, 205, 206, 474, 476, 503, 530, 552, 553
cuts, 227
direction, 264, 285, 291
displaying, 265
dual value, 227, 423
function, 57, 291, 585, 586
getting, 165, 167, 503, 504, 552
integrality, 69
length, 369
loading, 117, 476, 553
modifying, 207, 530
name, 168, 505
nonlinear data, 171
parameters, 72
primal value, 227
printing, 57
ranges, 218, 433, 540
row, 285
sense, 565
Visual Basic example, 291
operators, 337, 586
postfix notation, 339
optimal solution, 259, 287, 295, 494
optimality tolerance, 72, 73, 82
optimization, 139, 259, 421, 490
optimization method, LP, 80, 94
optimization routines, 139, 490
options, supported, 23
OR function, 343
order of precedence, 586
Ox statistical functions, 557
oxLINDO, 557

P

parameters, 42, 53, 99, 422, 568

getting, 45, 466, 467, 468
setting, 48, 466
parentheses, 338, 352, 586
Pareto distribution, 416
partial derivatives
calculating, 224, 226, 336
getting, 170, 171, 507
setting, 234, 549
partial pricing, 57
passing data, 276
password. See license key
Pearson correlation, 415
percent function, 339
PI, 339
piecewise linear, 578
plant location, 69, 227
plus, 586
Pluto Dogs, 280
Poisson, 341
Poisson distribution, 416
portfolio selection, 300, 560
positive definite, 295
positive semi-definite, 295, 584
postfix notation, 337, 368, 592
post-solving, 85
power function, 339
precedence order, 337, 352, 586
prefixes, 17
preprocessing, 59, 61, 73, 76, 81, 85
present value, 340, 343
pricing, 57
primal
infeasibility, 227, 423, 546
model, 29, 30
objective value, 227
simplex, 57, 60, 69, 77, 80, 94, 139, 265
solution, 54, 148, 150, 543
values, 151, 495
print level, 62, 76
printing objective value, 57
priorities, 136, 138, 486, 489
probability, 340
probing, 61, 76, 81
product form inverse, 2
product mix, 269
progress of solver, 421
protocol cdecl, 422
prototypes, 262
PSL, 340
PUSH instruction, 343, 346
put option, 413

Q

QMATRIX section, 296, 314
QSECTION, 296
quadratic constraint, 580
quadratic objective, 579
quadratic program, 200, 326, 479, 510, 511, 524, 573
constraints, 295
data, 173, 174
examples, 295
loading, 121, 125
MATLAB, 554
multistart, 353
quadratic programs as SOCP, 326
quadratic recognition, vii, 64
quadruplet entry for QC models, 297
QUADS, 296
query routines, 144
callback functions, 227, 546
errors, 566
MIP models, 287, 294
mxLINDO, 498
solver status, 422

R

radians, 340
random, 241
random number, 342
ranges
analysis, 215, 218, 433, 538
bounds, 214
names, 168, 505
vectors, 119, 477
rank correlation, 415
reading
LINDO format, 456
MATLAB, 555
models, 25
MPS format, 457
real bounds, 78
real numbers, 115, 474
Recourse Models, 391
reduced costs, 72, 81, 151, 496
reduced gradient solver, 59
reduction, 63
cuts, 227
dual, 61, 76, 81
of coefficients, 61, 69, 76, 81
refactorization, 53
reformulation, algebraic, 86
relative optimality tolerance, 72, 73
retrieving parameters, 42, 466
Reverse Polish notation, 337, 592
right-hand side
adding, 194, 518
arguments, 452

constraints, 264, 531, 586
getting, 127, 164, 165, 502, 503
increase/decrease, 215
loading, 118, 476
modifying, 207
names, 168, 505
values, 222
vector, 119, 125, 477
Visual Basic example, 275
rotated quadratic cone, 581
rounded solutions, 87, 588, 589
routines
auxiliary, 552
callback management, 227, 546
creation, 21
deletion, 21, 200, 203, 524, 526
errors, 566
memory management, 238
MIP models, 287, 294
model loading, 114, 473
model modification, 193, 517
mxLINDO, 452
optimization, 139, 490
query, 144, 160, 498
random number generation, 241
sampling routines, 245
solver status, 422
row
format, 194, 585
index vector, 11, 12
indices, 166, 176, 196, 276, 504, 513, 520
names, 119, 477
nonlinear, 120, 170, 478, 507
objective, 285
separable, 335
runlindo, 7
running an application, 267
runtime license, 92

S

sampl.c, 267
sampl.exe, 267, 268
sampl.obj, 268
Sample SP Problems, 411
samplec.mak, 267
samplevb.frm, 429
sampling routines, 245
SC bound type, 576
scaling, 56
scatter search, 352
Scenario Tree, 391
second order cone, 581
second-order cone
examples, 311
second-order-cone optimization, 311
selective constraint evaluation, 60
semi-continuous variable, 576
sense, of objective, 565
sensitivity analysis, 433
separable, 335
serial number, 23
setting parameters, 42, 48, 466
Setting up SP Models, 393
simple lower bound, 151
simple lower/upper bound, 587, 589
simplex method, 69, 77, 80, 94, 135, 486
dual, 57, 139, 265
iterations, 227
primal, 57, 139, 265
Simplex method, 2
sine, 340
size of version, 23, 91, 567, 568
slack values, 150, 152, 497
SLB, 587, 589
SLP pricing, 60, 137
smooth models, viii, 350, 354
SOCP, 311
MPS format, 314
SOCP Constraints, 322
SOCP Form, 321
Solaris, 8
solution, 259, 287, 494
analyzing, 433
dual, 148
incumbent, 83, 84, 227, 430
infeasible, 31, 216, 435, 460
nonoptimal, 588, 589
primal, 148, 150, 543
query routines, 144, 491
rounded, 588, 589
unbounded, 31, 435, 437, 460
writing, 34, 463
solver, 329
barrier, 53, 58, 60, 77, 80, 92, 94, 139, 265
branch-and-bound, 142, 230, 287, 293, 375, 491, 547
enumeration, 74
global solver, vii, 141, 354, 375, 490
initialization, 135, 486
interrupt, 53, 422, 430, 568
knapsack, 74
multistart, vii, 3
multistart solver, 371
nonlinear, vii, 62, 139, 295
progress, 421
quadratic, vii
solver status, 279, 422, 432, 433
type, 565
SOS, 577

SOS2 set, 578
sparse matrix representation, 10–12, 114, 329, 473
Spearman rank correlation, 415
Special Ordered Sets, 577
splitting lines, 586
square root, 339
stack based computer, 338
staffing model, 279
stage, 38, 39, 97, 110, 111, 128, 129, 130, 134, 154, 158, 180, 181, 186, 189, 191
standard Normal cdf, 340
standard Normal inverse, 343, 346
start, column, 11, 12, 285, 291
starting basis, 63, 135, 486
starting points, 116, 139, 353, 475, 487, 489
status of variables, 144, 147, 213
steepest edge pricing, 57, 60
stochastic information, 109
stochastic programming, 353, 389
Stochastic Programming, 389
stochastic solver, vii
storing data, 21
strong branching, 78
structure creation/deletion routines, 21, 452
student *t* distribution, 341
Student-t distribution, 416
SUB, 587, 589
subtraction, 339
successive linear programming, 3
sufficient set, 216, 217, 436, 437, 539, 540
summation, 342
supported options, 23
symmetric, 581
symmetric matrix., 298
syntax, 269, 585

T

t distribution, 341
tangent, 340
text format (ASCII), 26
thread safe, 421, 431
three vector representation, 11
time limit, 58, 70, 79, 80, 89, 91, 95, 568
title, 119, 168, 477, 505, 587, 589
tolerances, 55, 63, 72, 73, 77, 82, 83
transformation
  backward, 221, 541
  forward, 222, 542
trial license, 91
triangular distribution, 343
true, 340
types of constraints
  adding, 194, 518
  C++ example, 264
  errors, 565
  getting, 160, 162, 164, 165, 498, 500, 502, 503, 552
  loading, 118, 476, 553
  modifying, 528
types of cuts, 69, 70, 72
types of data, 17, 42, 263

U

unbounded, 31, 433, 435, 437, 460, 589
  MATLAB, 535, 540
unformatted MPS file, 26, 565
uniform distribution, 344
Uniform distribution, 416
unsupported features, 567
upper bounds
  adding, 196
  best, 212
  getting, 166
  LINDO files, 587
  loading, 118
  MATLAB, 475, 476, 503, 520, 552, 553
  MIPs, 71
  modifying, 209, 533
  MPS files, 573
  nonlinear programs, 117, 336
  objective, 167, 504
  SUB, 587, 589
  Visual Basic example, 276
upper triangle, 298
USER function, 342
user interface, 329, 421, 428
Usercalc(), 347
user-defined function, 381

V

value vector, 11
Value-At-Risk, 326
variables
  adding, 194, 195, 520
  artificial, 55, 63
  binary, 26, 167, 504, 587, 588
  block structure, 137
  bounded, 118, 166, 196
  bounded, MATLAB, 475, 476, 503, 520, 552, 553
  branch-and-bound, 139
  branching on, 78, 124, 486
  branching priorities, 136, 138, 489
  coefficients, 167, 504
  continuous, 139, 148, 158, 352, 589
  decision, 285, 291
  defining, 367

deleting, 203, 526
discrete, 352
displaying, 265
dual, 145, 492, 493
environment, 267
errors, 568
fixed, 72, 81, 573
free, 573, 587, 588
general integer, 167, 279, 504, 587, 588
getting, 167, 504
index of, 167
initial values, 139, 487, 489
integer, 148, 158, 573
integer feasible tolerance, 73, 77
internal index, 178, 179, 203, 514, 515, 526
left-hand sides, 587
limit, 91
loading, 482, 483, 484, 485
long, 269
MIPs, 279, 287
modifying, 209
name hashing, 238
names, 119, 125, 168, 178, 179, 505, 514, 515, 573, 585
negative, 573, 587
nonlinear, 91, 120, 170, 172, 478, 507
number of, 115, 117, 473, 474, 476, 503
primal, 151, 495
priorities, 136
quadratic, 121, 197
reduced costs, 151, 152, 496
slack/surplus values, 55, 63, 150, 152, 497
splitting lines, 586
status, 144, 147, 213
types of, 123, 124, 167, 180, 481, 482, 484, 485, 504, 516, 533
values, 277
variance reduction, 416
VB, 277
VB modules, 429
vcvars32.bat, 267
Vector AND, 343
Vector OR, 343
vector Push, 345
vectors, 11, 12, 119, 125, 221, 329, 337, 477, 520
versions, 23, 58, 91, 297, 315, 567, 568
violated constraints, 55, 63
Visual Basic, 233, 235
Visual Basic example, 269, 287, 429
Visual Basic for Applications, 429
Visual C++ 6, 266
Visual C++ example, 280

W

warm start, 239, 240, 353, *See also* initial values
Weibull distribution, 416
wizard, 282
wrap function, 341
wrapping, 286, 292
writing
dual, 458, 459
LINDO format, 461, 586
LINGO format, 461
models, 25
MPS format, 458, 462, 463
solutions, 34, 463